π Framework

Christiano Braga
cbraga@ic.uff.br

Instituto de Computação
Universidade Federal Fluminense

May 22, 2018

Abstract. π framework is comprised by π-lib is a library of common programming language constructs whose semantics are formally specified and π-automata, an automata-based formalism to describe the operational semantics of programming languages. To give semantics to a programming language means to relate constructions of the given language with the constructs of π-lib. It is implemented in the Maude language.

1 Introduction

The π framework is comprised by π-lib, a set of programming languages constructs inspired by Peter Mosses' Component-Based Semantics [12] and π-automata, an automata-based formalism to describe the operational semantics of programming languages, that generalizes Gordon Plotkin's Interpreting Automata approach [14]. Since π-lib has an automata-based semantics, π-lib programs can be model checked or be subjected to any automata-based validation technique. The Maude [7] implementation of π-lib[1] allows for rewriting-based execution, Linear Temporal Logic model checking and narrowing-based symbolic execution of π-lib programs.

This report describes π, an automata-based semantic framework for teaching formal compiler construction and its implementation in the Maude language. From a pedagogical perspective, being automata-based helps closing the gap between compiler construction and subjects such as Introduction to Programming Languages, Programming Languages Semantics and Formal Languages and Automata Theory, that usually precedes Compiler Construction courses. Our previous experience with the formal specification of programming languages, in particular the component-based framework, helped us devise an approach of operational character that formalizes a core (or intermediate) language amenable to automated verification and code generation.

The remainder of this report is organized as follows. Section 2 recalls some preliminary material to the discussion of π-automata, subject of Section 3. The π-automata semantics of π-lib is discussed in Section 4.1, together with its Maude implementation and the IMP compiler.

---

[1] http://github.com/ChristianoBraga/BPLC

2 Preliminaries

Before introducing the π framework and its Maude implementation we first recall, in Section 2.1, some basic concepts from operational semantics and model checking and, in Section 2.2, the Maude language.

2.1 Transition systems, structural operational semantics and model checking

This Section recalls, very briefly, just for completeness, the basic concepts of labeled and unlabeled transition systems, structural operational semantics and model checking.

A labeled transition system [1] (LTS) is a tuple $\mathscr{T} = (S, \rightarrow, L)$, where S denotes the set of the states of the system, $\rightarrow \subseteq S \times L \times S$ is the transition relation, with L a set of labels. An unlabeled transition system is an LTS where $L = \emptyset$. A concurrent system may have labeled transition systems as models, with L denoting the set of actions of the system, that may eventually synchronize.

Labeled transition systems are the standard models of structural operational semantics descriptions. Given an SOS description $\mathscr{M} = (G, R)$ specifying the semantics of a programming language L, the set G defines the grammar of L while relation R represents the semantics of L (either static or dynamic) in a syntax-directed way. Equation 1 presents the general form of the (unlabeled) rule for the inductive step of the evaluation of a programming language construct f in the SOS framework, where ρ and ρ' are environments, ρ' is the result of some computation involving ρ, f is programming language construct and t_i its parameters, $\sigma, \sigma', \sigma_i, \sigma'_i$ are memory stores, with σ' the result of some computation of $\uplus_{i=1}^n \sigma'_i$, and Cnd is a predicate not involving transitions.

$$\frac{\rho' \vdash \bigwedge_{i=1}^n t_i, \sigma_i \Rightarrow t'_i, \sigma'_i}{\rho \vdash f(t_1, t_2, \ldots, t_n), \sigma \Rightarrow f(t'_1, t'_2, \ldots, t'_n), \sigma'} \text{ if } Cnd \tag{1}$$

Typically, SOS rules have a *sequent* in the conclusion of the form $\rho \vdash g, \sigma \Rightarrow g', \sigma'$, where g and g' are derivations of G. If one looks at the conclusion as a transition of the form $(\rho, g, \sigma) \Rightarrow (\rho, g', \sigma')$ than the construction of the (unlabeled) transition system $\mathscr{T}$ from $\mathscr{M}$ becomes straightforward with $(\rho, g, \sigma) \in S$.

Model checking [5] is an automata-based automated validation technique to solve the "question" $\mathscr{T}, s \models \varphi$, that is, does model $\mathscr{T}$, a transition system, or Kripke structure in Modal Logic [9] jargon, with initial state s, satisfies property φ? The standard algorithm checks if the language accepted by the intersection Büchi automaton (a regular Ω-automaton, that is, an automaton that accepts infinite words) of $\mathscr{T}$ and $\neg\varphi$ is empty.

2.2 Maude

In this Section we introduce the main elements of the Maude language, our choice of programming language for this work.

The Maude system and language [7] is a high-performance implementation of Rewriting Logic [11], a formalism for the specification of concurrent systems that has been shown to be able to represent quite naturally many logical and semantic frameworks [10].

Maude[2] is an algebraic programming language. A program in Maude is organized by modules, and every module has an initial algebra [8] semantics. Module inclusion may occur in one of three different *modes*: `including`, `extending` and `protecting`. The `including` mode is the most liberal one and imposes no constraints on the preservation of the algebra of the included module into the including one, that is, both "junk" and "confusion"[3] may be added. Inclusion in `extending` mode may add "junk" but no "confusion", while inclusion in `protecting` mode adds no "junk" and no "confusion" to the included algebra. Module inclusion is not enforced by the Maude engine, being understood only as an indication of the intended inclusion semantics. Such declarations, however, are part of the semantics of the module hierarchy and may be important for Maude-based tools, such as a theorem prover for Maude specifications, that would have to discharge the proof obligations generated by such declarations.

Computations in Maude are represented by rewrites according to either equations, rules or both in a given module. Functional modules may only declare equations while system modules may declare both equations and rules. Equations are assumed (that is, yield proof-obligations) to be Chruch-Rosser and terminating [2]. Rules have to be coherent: no rewrite should be missed by alternating between the application of rules and equations. A (concurrent) system is specified by a rewrite system $\mathscr{R} = (\Sigma, E \cup A, R)$ where Σ denotes its signature, E the set of equations, A a set of axioms, and R the set of rules. The equational theory $(\Sigma, E \cup A)$ specifies the states of the system, t terms in the Σ-algebra modulo the set of E equations plus A-axioms such as associativity, commutativity and identity. Combinations of such axioms give rise to different rewrite theories such that rewriting takes place modulo such axioms. Rules R specify the (possibly) non-terminating behavior, that takes place modulo the equational theory $(\Sigma, E \cup A)$. Another interesting feature of Maude is to support *non-linear patterns* (when the same variable appears more than once in a pattern) both in equations and rules. Section 3.3 exemplifies how this feature is intensively used in the Maude implementation of the π-automata framework.

An interesting remark regards the decision between modeling behavior as equations or rules. One may specify (terminating) system behavior with equations. The choice between equations and rules provides an *observability gauge*. In the context of a software architecture, for instance, *non-observable* (terminating) actions, internal to a given component, may be specified by equations, while *observable* actions, that relate components in a software architecture, may be specified as rules. Sec-

---

[2] Maude allows for programming with different Equational Logics: Many-sorted, Order-sorted or Membership Equational Logic. In this paper, Maude programs are described using Order-sorted Equational Logics.

[3] Informally, when "junk" may be added to an algebra but "confusion" may not, as in `extending` mode, it means that new terms may be included but are not identified with old ones.

tion [3.3](#) illustrates how this "gauge" is used in the Maude implementation of the π framework.

A compiler can be implemented in Maude as a *meta-level* application. Such a Maude application uses the so called *descent functions* [6, Ch.11] that represent modules as terms in an *universal* theory, implemented in Maude as a system module called META-LEVEL. Some of the descent functions are metaParse, metaReduce, metaRewrite and metaSearch. Function metaParse receives a (meta-represented) module denoting a grammar, a set of quoted identifiers representing the (user) input and a quoted identifier representing the rule that should be applied to the given input qids, and returns a term in the signature of the given module. Descent function metaReduce receives a (meta-represented) module and a (meta-represented) term and returns the (meta-represented) canonical form of the given term by the exhaustive application of the (Church-Rosser and terminating) *equations*, only, of the given module. A interesting example of metaReduce is the invocation of the model checker at the metalevel: (i) first, module MODEL-CHECKER must be included in a module that also includes the Maude description of the system to be analyzed, and (ii) one may invoke metaReduce of a meta-representation of a term that is a call to function modelCheck, with appropriate parameters, defined in module MODEL-CHECKER. Finally, function metaRewrite simplifies, in a certain number of steps, a given term according to both equations and rules (assumed coherent, that is, no term is missed by the alternate application of equations and rules) of the given module. Meta-search looks for a term that matches a given *pattern*, from a given term, according to a choice of rewrite relation from $\Rightarrow^*$, $\Rightarrow^+$, $\Rightarrow^!$, denoting the reflexive-transitive closure of the rewrite relation, the transitive closure of the rewrite relation or the rewrite relation that produces only canonical forms.

Section [3](#) continues this paper by introducing π-automata.

3 π-automata

3.1 Interpreting automata

In [14], Plotkin defines the concept of Interpreting Automata as finite-state Transition Systems as a semantic framework for the operational semantics of programming languages. Interpreting Automata are now recalled from the perspective of Automata Theory.

Let $\mathscr{L}$ be a programming language accepted by a Context Free Grammar (CFG) $G = (V, T, P, S)$ defined in the standard way where V is the finite set of variables (or non-terminals), T is the set of terminals, $P \subseteq V \times (V \cup T)^*$ and $S \notin V$ is the start symbol of G. An Interpreting Automaton for $\mathscr{L}$ is a tuple $\mathscr{I} = (\Sigma, \Gamma, \rightarrow, \gamma_0, F)$ where $\Sigma = T$, Γ is the finite set of configurations, $\rightarrow \subseteq \Gamma \times \Gamma$ is the transition relation, $\gamma_0 \in \Gamma$ is initial configuration, and F the finite set of final configurations. Configurations in Γ are triples of the general form

$$\Gamma = \textit{Value Stack} \times \textit{Memory} \times \textit{Control Stack}.$$

where *Value Stack* $= L(G)$ with $L(G)$ the language accepted by G, the set *Memory* is a finite map *Var* $\rightarrow_{fin}$ *Storable* with *Var* $\in V$ and *Storable* $\subset T^*$, and the elements of the

Control Stack $= L(G) \cup KW$, where *KW* is the set of keywords of $\mathscr{L}$. A computation in $\mathscr{I}$ is defined as $\rightarrow^*$, the reflexive-transitive closure of the transition relation.

As an example, let us consider a programming language $\mathscr{L}$ with arithmetic expressions, Boolean expressions and commands with the CFG in Figure 1.

$$
\begin{aligned}
\textit{Prog} &::= \textit{ComSeq} \\
\textit{ComSeq} &::= \texttt{nop} \mid \textit{Com} \mid \textit{Com} \,;\, \textit{ComSeq} \\
\textit{Com} &::= \textit{Var} := \textit{Exp} \mid \\
&\qquad \texttt{if } \textit{BExp ComSeq ComSeq} \mid \\
&\qquad \texttt{while } \textit{BExp ComSeq} \\
\textit{Exp} &::= \textit{BExp} \mid \textit{AExp} \\
\textit{BExp} &::= \textit{Exp BOP Exp} \\
\textit{BOP} &::= \, = \mid \texttt{or} \mid \sim \\
\textit{AExp} &::= \textit{AExp AOP AExp} \\
\textit{AOP} &::= \, + \mid - \mid *
\end{aligned}
$$

Fig. 1. CFG for $\mathscr{L}$

The values in the *Value Stack* are elements of the set $\{\mathbb{T} \cup \mathbb{N} \cup \textit{Var} \cup \textit{BExp} \cup \textit{Com}$, where $\mathbb{T}$ is the set of Boolean values, $\mathbb{N}$ is the set of Natural numbers, with *Var* the set of variables, *BExp* the set of Boolean expressions, and *Com* the set of commands of $\mathscr{L}$. The *Control Stack* is defined as the set $(\textit{Com} \cup \textit{BExp} \cup \textit{AExp} \cup \textit{KW})^*$, where *AExp* is the set of arithmetic expressions and $KW = \{+, -, *, =, \texttt{or}, \sim, :=, \texttt{if}, \texttt{while}, ;\}$.

Informally, the computations of an Interpreting Automaton mimic the behavior of a calculator in Łukasiewicz postfixed notation, also known as reverse polish notation. A typical computation of an Interpreting Automaton *interprets* a statement $c(p_1, p_2, \ldots, p_n) \in L(G)$ on the top of *Control Stack* C of a configuration $\gamma = (S, M, C)$, by unfolding its subtrees $p_i \in L(G)$ and $c \in KW$ that are then pushed back into C, and possibly updating the *Value Stack S* with intermediary results of the interpretation of the $c(p_1, p_2, p_n)$, and the *Memory*, should $c(p_1, p_2, p_n) \in L(\textit{Com})$.

For the transition relation of $\mathscr{I}$, let us consider the rules for arithmetic sum expressions in Figure 2, where e_i are metavariables for arithmetic expressions, and

$$
\langle S, M, n\ C \rangle \Rightarrow \langle n\ S, M, C \rangle \tag{2}
$$

$$
\langle S, M, (e_1 + e_2)\ C \rangle \Rightarrow \langle S, M, e_1\ e_2 + C \rangle \tag{3}
$$

$$
\langle n\ m\ S, M, + C \rangle \Rightarrow \langle (n+m)\ S, M, C \rangle \tag{4}
$$

Fig. 2. Rules for addition in Interpreting Automata

$n, m \in \mathbb{N}$. Rule 3 specifies that when the arithmetic expression $e_1 + e_2$ is on top of the control stack C, then its operands should be pushed to C and then the operator

+. Operands e_1 and e_2 will be recursively evaluated, as a computation is the reflexive-transitive closure of relation $\rightarrow$, leading to a configuration with an element in $\mathbb{T} \cup \mathbb{N}$ left on top of the value stack S, as specified by Rule 2. Finally, when + is on top of the control stack C, and there are two Natural numbers on top of S, they are popped, added and pushed back to the top of S.

Finally, there is one quite interesting characteristic of Interpreting Automata: *transitions do not appear in the conditions of the rules*, a characteristic that can be quite desirable from a proof theoretic standpoint, in particular in the context of term rewriting systems (see Section 3.3), as pointed out by Viry [16] and later by Roşu in [15], for instance. As opposed to transition rules that admit transitions in its premises, as in the Structural Operational Semantics (SOS) framework, for instance, also defined in [14], Interpreting Automata evaluation uses the control stack.

As an example, let us recall Equation 1, the general form of the rule for the inductive step of the evaluation of a programming language construct f in the SOS framework,

$$\frac{\rho' \vdash \bigwedge_{i=1}^n t_i, \sigma_i \Rightarrow t_i', \sigma_i'}{\rho \vdash f(t_1, t_2, \ldots, t_n), \sigma \Rightarrow f(t_1', t_2', \ldots, t_n'), \sigma'} \text{ if } Cnd$$

where ρ and ρ' are environments, ρ' is the result of some computation involving ρ, f is programming language construct and t_i its parameters, $\sigma, \sigma', \sigma_i, \sigma_i'$ are memory stores, with σ' the result of some computation of σ_i', and Cnd is a predicate not involving transitions. The Interpreting Automata rule for Rule 1 is as follows,

$$\{f(t_1, t_2, \ldots, t_n) \ C, \rho, \sigma, \ldots\} \Rightarrow \{t_1 t_2 \ldots t_n f \ C, \rho, \sigma, \ldots\} \text{ if } Cnd, \tag{5}$$

where C is the control stack. Note that induction will take care of evaluating a t_i when it is on top of the control stack, so there is no need to explicitly require transitions of the form $t_i \Rightarrow t_i'$ as premises or conditions to Rule 5. Copies of the environment (such as ρ' in Rule 1) and side-effects are *naturally* calculated during the computation process by the application of the appropriate rule for the term on top of the control stack.

3.2 π-automata

π-automata are Interpreting Automata whose *configurations are sets of semantic components* that include, at least, a *Value Stack*, a *Memory* and a *Control Stack*. Plotkin's stacks and memory in Interpreting Automata (or environment and stores of Structural Operational Semantics) are generalized to the concept of *semantic component*, as proposed by Peter Mosses in Modular SOS approach to the formal semantics of programming languages.

Formally, a π-automaton is an Interpreting Automaton where, given an abstract finite set *Sem*, for semantics components $\Gamma = \uplus_{i=1}^n Sem$, with $n \in \mathbb{N}$, $\uplus$ denoting the disjoint union operation of n semantic components, with *Value Stack*, *Memory* and *Control Stack* subsets of *Sem*.

The semantic rules for sum in π-automata look very similar to the rules in Fig. 2. For the rules in Fig. 3 "…" [4] are adopted as notation for "don't care" semantic com-

$$\{S, M, n\ C, \ldots\} \Rightarrow \{n\ S, M, C, \ldots\} \tag{6}$$

$$\{S, M, (e_1 + e_2)\ C, \ldots\} \Rightarrow \{S, M, e_1\ e_2 + C, \ldots\} \tag{7}$$

$$\{n\ m\ S, M, + C, \ldots\} \Rightarrow \{(n + m)\ S, M, C, \ldots\} \tag{8}$$

Fig. 3. Rules for addition in π-automata

ponents, that is, those components that are not relevant for the specification of the semantics of a particular language construct.

The point is that if one wants to extend one's Interpreting Automata specification with new semantic components, say disjointly uniting an output component (representing standard output in the C language, for instance), understood as a sequence of values, to the already existing disjoint set of environments and stores, would require a reformulation of the existing specification. For instance, the specification in Fig. 2 would require such reformulation while in Fig. 3 it would not. The rules in the latter have the "don't care" variable that matches any, or no component at all, that may be together with S, M and C. Another way of (informally) understanding patterns of the form $\{\ldots, S, \ldots\}$ is to think about it as a projection function for the S component out of a configuration. Semantic component composition is *monotonic*, as the addition of new semantic components does not affect the transition relation, that is, $x \Rightarrow y$ implies $f(x) \Rightarrow f(y)$, where $x, y \in \Gamma$ and f is a function that adds a new semantic component to Γ.

3.3 π-automata and Term Rewriting

A π-automaton $\mathscr{I} = (\Sigma, \Gamma, \rightarrow, \gamma_0, F)$ can be seen as an unlabeled Transition System $\mathscr{I} = (\Gamma, \rightarrow)$ and therefore a Term Rewriting System [2] when the latter is understood as $\mathscr{T} = (A, \longrightarrow)$ where A is a set and $\longrightarrow$ a reduction relation on A. Clearly, the set of configurations Γ is A and the transition relation of the Interpreting Automata is the reduction relation of the Term Rewriting System.

There is an interesting point on the relation between the semantics of a programming language construct, specified by a π-automata, and the *properties* that one may require from the reduction relation of the associated Term Rewriting System. Let us first recall two basic properties of a reduction relation from [2, Def.2.1.3],

- Church-Rosser: $x \overset{*}{\longleftrightarrow} y \implies x \downarrow y$,
- termination: there is no infinite reduction $a_0 \rightarrow a_1 \rightarrow \ldots$.

where $x, y \in A$, $\overset{*}{\longleftrightarrow}$ denotes the reflexive-transitive-symmetric closure of $\longrightarrow$, and $x \downarrow y$ denotes that x and y are joinable, that is, $\exists z, x \overset{*}{\longrightarrow} z \overset{*}{\longleftarrow} y$. In rewriting modulo equational theories [2, Ch. 11], otherwise non-terminating systems become terminating when an algebraic property, such as commutativity, is incorporated into the

---

[4] This notation is similar to the one defined by Chalub and Mosses in the Modular SOS Description Formalism, which is implemented in the Maude MSOS Tool [4].

rewriting process. Given a Term Rewriting System $(A, \longrightarrow)$, let E be a set with the identities induced by a given property, such as commutativity, and R the remaining identities induced by $\longrightarrow$. Rewriting then occurs on equivalence classes of terms, giving rise to a new relation, $\longrightarrow_{R/E}$, defined as follows:

$$[s]_{\approx E} \longrightarrow_{R/E} [t]_{\approx E} \Leftrightarrow \exists s', t'.s \approx_E s' \longrightarrow_R t' \approx_E t.$$

Moving back to π-automata, the semantics of a programming language construct $c(p_1, \ldots, p_n)$ is *functional*, where c is the construct and p_i its parameters, when given any configuration $\gamma = \{c(p_1, \ldots, p_n) \ C, \ldots\}$, there exists a single γ' such that $\gamma \Rightarrow^* \gamma'$ and the computation is finite. The semantics of a programming language construct $c(p_1, \ldots, p_n)$ is *relational* when given any configuration $\gamma = \{c(p_1, \ldots, p_n) \ C, \ldots\}$, where $C \in Control\ Stack$, the computations starting in γ may lead to different γ'_i and may not terminate.

Therefore, if the semantics of a programming language construct is functional, one must require the associated reduction relation to be Church-Rosser and terminating. No constraints are imposed to the reduction relation when the semantics is relational.

As an illustration, according to this definition, the semantics of addition is *functional* but an undefined loop (such as a while command) semantics is *relational* as its execution may not terminate.

In order to support the specification of monotonic rules in a modular way, one last thing is required from the Term Rewriting System associated with a π-automata: rewriting modulo associativity, idempotence and commutativity. In other words, *set*-rewriting takes place, not simply term rewriting, while representing π-automata as Term Rewriting Systems, as each rule rewrites a set of semantic components.

π-automata in Maude Maude parameterized programming capabilities are used to implement π-automata. The main datatype of π-automata is Generalized SMC (GSMC in Listing 1), a set of semantic components. The trivial view SemComp maps terms of sort Elt to terms of sort SemComp. Module GSMC then imports the Maude predefined module SET parameterized by view SemComp, of semantic components, implemented in Maude by functional module GSMC-SORTS. A configuration of a π-automaton is declared with constructor `<_> : SetSemComp -> Conf` that gives rise to terms such as `< ` c_1 `, ` c_2 ` >` where c_i is a semantic component. (Keyword `format` is an attribute to display colored terms in a terminal while running Maude.)

```
1  fmod SEMANTIC−COMPONENTS is
2      sorts SemComp .
3  endfm
4
5  view SemComp from TRIV to SEMANTIC−COMPONENTS is
6      sort Elt to SemComp .
7  endv
8
9  fmod GSMC is
10     ex VALUE−STACK .
11     ex MEMORY .
12     ex CONTROL−STACK .
13     ex ENV .
14     ex SET{SemComp} ∗ (op empty to noSemComp) .
```

```
15
16    sorts Attrib Conf EnvAttrib StoreAttrib ControlAttrib ValueAttrib .
17    subsort EnvAttrib StoreAttrib ControlAttrib ValueAttrib < Attrib .
18    op <_> : Set{SemComp} −> Conf [format(c! c! o)] .
19
20    −−− Semantic components
21    op env : −> EnvAttrib .
22    op sto : −> StoreAttrib .
23    op cnt : −> ControlAttrib .
24    op val : −> ValueAttrib .
25    op _:_ : EnvAttrib Env −> SemComp [ctor format(c! b! o o)] .
26    op _:_ : StoreAttrib Store −> SemComp [ctor format(r! b! o o)] .
27    op _:_ : ControlAttrib ControlStack −> SemComp [ctor format(c! b! o o)] .
28    op _:_ : ValueAttrib ValueStack −> SemComp [ctor format(c! b! o o)] .
29 endfm
```

Listing 1. Generalized SMC in Maude

Recall that the elements of the disjoint union $\biguplus_{i=1}^{n} Sem$ are ordered pairs (s, i) such that i serves as an index indicating which semantic component s came from. This is exemplified in Maude with the memory store component in Listing 2. The constructor operator `sto` functions as the index for the memory store component and the constructor operator `_:_` to represent ordered pairs (s, i) where s is the memory store.

```
1    op sto : −> StoreAttrib [ctor] .
2    op _:_ : StoreAttrib Store −> SemComp [ctor format(r! b! o o)] .
```

Listing 2. Memory store semantic component in Maude

Now, for the transition rules, they are represented either by equations or rules, depending on the semantic character of the programming language construct being formalized. In the case of arithmetic expressions, in Listing 3, their character is functional and therefore are implemented as equations in Maude. For sum, in equation `add-exp1`[5] first operands `E1:Exp` and `E2:Exp` are unfolded, and then pushed back to the control stack `C`, together with `ADD`, an element of set *KW*. (Recall that *Control Stack* = $(Com \cup BExp \cup AExp \cup KW)^*$.) Equation `add-exp2` implements the case where both `E1:Exp` and `E2:Exp` have been both evaluated and their associated (Rational) value (in this implementation) was pushed to the value stack. When `ADD` is on top of the control stack then the two top-most values in the value stack are added. (Note that + symbol in `add-exp2` denotes sum in the Rationals whereas in `add-exp2` is the symbol for sum in language $\mathscr{L}$.)

```
1 eq [add−exp1] : < cnt : (E1:Exp + E2:Exp) C:ControlStack), ... > =
2                 < cnt : (E1:Exp E2:Exp ADD C:ControlStack), ... > [variant] .
3
4 eq [add−exp2] : < cnt : (ADD C:ControlStack,
5                 val : (val(R1:Rat) val(R2:Rat) SK:ValueStack), ... > =
6                 < cnt : C:ControlStack,
7                 val : (val(R1:Rat + R2:Rat) SK:ValueStack), ... > [variant] .
```

Listing 3. π-automaton sum rules in Maude

[5] Keyword `variant` is an attribute for equations and means that the given equation should be used in the variant unification process. Due to space constraints, this feature is not discussed in this paper. The keyword is left in the code snippet to present the actual executable code for the tool.

3.4 Model checking π-automata

Model checking (e.g. [5]) is perhaps the most popular formal method for the validation of concurrent systems. The fact that is an *automata-based automated validation technique* makes it a nice candidate to join a simple framework for teaching language construction, such as the one proposed in this paper, that also aims at validation.

This Section recalls the syntax and semantics for Linear Temporal Logic, one the Modal Logics used in model checking, and discusses how to use this technique to validate π-automata.

The syntax of Linear Temporal Logic is given by the following grammar

$$\phi ::= \top \mid \bot \mid p \mid \neg(\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \to \phi) \mid$$
$$(\mathscr{X}\phi) \mid (\mathscr{F}\phi) \mid (\mathscr{G}\phi) \mid (\phi\mathscr{U}\phi)$$

where connectives $\mathscr{X}, \mathscr{F}, \mathscr{G}$, and $\mathscr{U}$, are called *temporal modalities*. They denote "neXt", "Future state", "Globally (all future states)", and "Until". There is a precedence among them given by: first unary modalities, in the following order $\neg, \mathscr{X}, \mathscr{F}$ and $\mathscr{G}$, then binary modalities, in the following order, $\mathscr{U}, \wedge, \vee$ and $\to$.

The standard models for Modal Logics (e.g. [9]) are Kripke structures, triples $\mathscr{K} = (W, R, L)$ where W is a set of worlds, $R \subseteq W \times W$ is the world accessibility relation and $L : W \to 2^{AP}$ is the labeling function that associates to a world a set of atomic propositions that hold in the given world. Depending on the modalities (or operators in the logic) and the properties of R, different Modal Logics arise such as Linear Temporal Logic. A *path* in a Kripke structure $\mathscr{K}$ represents a possible (infinite) scenario (or computation) of a system in terms of its states. The path $\tau = s_1 \to s_2 \to \dots$ is an example. A *sufix* of τ denoted τ^i is a sequence of states starting in i-th state. Let $\mathscr{K} = (W, R, L)$ be a Kripke structure and $\tau = s_1 \to \dots$ a path in $\mathscr{K}$. Satisfaction of an LTL formula ϕ in a path τ, denoted $\tau \models \varphi$ is defined in Figure 4.

$$\tau \models \top,$$
$$\tau \not\models \bot,$$
$$\tau \models p \ \textit{iff} \ p \in L(s_1),$$
$$\tau \models \neg\phi \ \textit{iff} \ \tau \not\models \phi,$$
$$\tau \models \phi_1 \wedge \phi_2 \ \text{iff} \ \tau \models \phi_1 \ \text{and} \ \tau \models \phi_2,$$
$$\tau \models \phi_1 \vee \phi_2 \ \text{iff} \ \tau \models \phi_1 \ \text{or} \ \tau \models \phi_2,$$
$$\tau \models \phi_1 \to \phi_2 \ \text{iff} \ \tau \models \phi_2 \ \text{whenever} \ \tau \models \phi_1,$$
$$\tau \models \mathscr{X}\phi \ \text{iff} \ \tau^2 \models \phi,$$
$$\tau \models \mathscr{G}\phi \ \text{iff for all} \ i \geq 1, \tau^i \models \phi,$$
$$\tau \models \mathscr{F}\phi \ \text{iff there is some} \ i \geq 1, \tau^i \models \phi,$$
$$\tau \models \phi\mathscr{U}\psi \ \text{iff}$$
$$\quad \text{there is some} \ i \geq 1 \ \text{such that} \ \tau^i \models \psi \ \text{and}$$
$$\quad \text{for all} \ j \in \{1, i-1\}, \tau^j \models \phi.$$

Fig. 4. Satisfaction relation of LTL formulae

A π-automata, when understood as a Transition System, is also a *frame*, that is, $\mathscr{F} = (W, R)$, where W is the set of worlds and R the accessibility relation. A Kripke

structure is defined from a frame representing a π-automata by declaring the labeling function with the following state proposition scheme:

$$\forall \sigma \in Memory, v \in Index(\sigma), r \in Storable,$$
$$\langle \sigma, \ldots \rangle \models p_v(r) =_{def} (\sigma(v) = r), \tag{9}$$

meaning that for every variable v in the index of the memory store component (which is a necessary semantic component) there exists a unary proposition p_v that holds in every state where v is bound to p_v's parameter in the memory store. A *poetic license* is taken here and π-automata, from now on, refers to the pair composed by a π-automata and its state propositions. As an illustrative specification, the LTL formula $\mathscr{G} \neg (p_1(crit) \wedge p_2(crit))$ specifies safety when p_i are state proposition formulae denoting the states of two processes and *crit* is a constant denoting that a given process is in the critical section, and formula $\mathscr{G}[p_1(try) \rightarrow \mathscr{F}(p_1(crit))]$ specifies liveness by stating that if a process tries to enter the critical section it will eventually will.

4 π-lib: Basic Programming Language Constructs

π-lib is a subset of Constructive MSOS [13], as implemented in [3, Ch. 6]. In Section 4.1, π-lib constructions are presented, their π-automata Semantics is discussed in Section 4.2 and a simple compiler for the IMP language is Maude, using π-lib, is described in Section 4.3.

4.1 π-lib signature

The signature of π-lib is organized in five parts, and implemented in four different modules in Maude: (i) Expressions, that include basic values (such as Rational numbers and Boolean values), identifiers, arithmetic and Boolean operations, (ii) Commands, statements that produce side effects to the memory store, (iii) Declarations, which are statements that construct the constant environment, (iv) output and (v) Abnormal termination.

Due to space constraints, only π-lib signature for expressions are discussed (see Listing 4). The remaining declarations follow a similar pattern. First, it includes modules QID, RAT, and GSMC, for quoted identifiers, rational numbers and Generalized SMC machines, respectively. Modules QID and RAT are part of Maude standard prelude while GMSC was defined in Listing 1. Next, module EXP declares sorts Exp, BExp and AExp, for (general) expressions, Boolean expressions and arithmetic expressions. Identifiers are subsorts of both Boolean expressions and arithmetic expressions, which are in turn subsorts of expressions. The latter are included in Control. Operator idn constructs Identifiers from Maude built-in quoted identifiers. Both arithmetic and boolean operations alike are declared as Maude operators, and so are elements of set *KW*.

```
1  fmod EXP is
2      pr QID . pr RAT .
3      pr GSMC .
4
```

```
5     sorts Exp BExp AExp .
6     subsort Id < BExp AExp < Exp < Control .
7
8     --- Identifiers
9     op idn : Qid -> Id [ctor format(!g o)] .
10
11    --- Arithmetic
12    op rat : Rat -> AExp [ctor format(!g o)] .
13    op add : AExp AExp -> AExp [format(! o)] .
14    op sub : AExp AExp -> AExp [format(! o)] .
15    op mul : AExp AExp -> AExp [format(! o)] .
16    op div : AExp AExp -> AExp [format(! o)] .
17
18    ops ADD SUB MUL DIV : -> Control [ctor] .
19
20    --- Boolean expressions
21    op boo : Bool -> AExp [ctor format(!g o)] .
22    op gt : Exp Exp -> BExp [format(! o)] .
23    op ge : Exp Exp -> BExp [format(! o)] .
24    op lt : Exp Exp -> BExp [format(! o)] .
25    op le : Exp Exp -> BExp [format(! o)] .
26    op eq : Exp Exp -> BExp [format(! o)] .
27    op neg : BExp -> BExp [format(! o)] .
28    op and : BExp BExp -> BExp [format(! o)] .
29    op or : BExp BExp -> BExp [format(! o)] .
30
31    ops LT LE EQ NEG AND OR : -> Control [ctor] .
32    ...
33 endfm
```

Listing 4. Signature for π-lib expressions in Maude

Listing 5 details the signature for π-lib commands, that is, statements that produce side-effects. The module CMD requires expressions, preserving its initial model, declares a nop command, that produces no effect on the the memory, a non-deterministic choice command, an assignment, a loop and a conditional. Since assignment, loop and if require the recursive evaluation of operands, they also declare constants in *KW*.

```
1  mod CMD is
2      pr EXP .
3
4      sort Cmd .
5      subsort Cmd < Control .
6
7      op nop : -> Cmd [ctor format(! o)] .
8      op choice : Cmd Cmd -> Cmd [ctor assoc comm format(! o)] .
9      op assign : Id Exp -> Cmd [ctor format(! o)] .
10     op loop : Exp Cmd -> Cmd [ctor format(! o)] .
11     op if : Exp Cmd Cmd -> Cmd [ctor format(! o)] .
12
13     ops ASSIGN LOOP IF : -> Control [ctor] .
14     ...
15 endfm
```

Listing 5. Signature for π-lib commands in Maude

Declarations are statements whose semantics affect the environment. π-lib declarations in Maude are declared in module DEC in Listing 6. Constants, references to memory locations, procedures, block declarations and operation calls form the statements in DEC. There are also elements of the signature for the the declaration of formal parameters, actual parameters in function declaration and call, respectivelly. Module DEC also declares a new semantic component called locs. It is used to

record the location that a reference allocates. Upon conclusion, the evaluation of a block frees all the locations allocated during its execution. (Listing 11 in Section 4.2 gives more details on the semantics of block evaluation.)

```
1  mod DEC is
2      ex CMD .
3
4      sorts Abs Blk Dec Formal Formals Actual Actuals LocsAttrib .
5      subsort Actuals Dec < Control .
6      subsort Formal < Formals .
7      subsort Exp < Actual < Actuals .
8      subsort Blk < Cmd .
9      subsort Abs < Bindable .
10
11     op cns : Id Exp −> Dec [ctor format(! o)] .
12     op ref : Id Exp −> Dec [ctor format(! o)] .
13     op prc : Id Blk −> Dec [ctor format(! o)] .
14     op prc : Id Formals Blk −> Dec [ctor format(! o)] .
15     op par : Id −> Formal [ctor format(! o)] .
16     op vod : −> Formal [ctor format(! o)] .
17     op for : Formals Formals −> Formals [ctor assoc format(! o)] .
18     op dec : Dec Dec −> Dec [ctor format(! o)] .
19     op blk : Cmd −> Blk [ctor format(! o)] .
20     op blk : Dec Cmd −> Blk [ctor format(! o)] .
21     op cal : Id −> Cmd [ctor format(! o)] .
22     op cal : Id Actuals −> Cmd [ctor format(! o)] .
23     op act : Actuals Actuals −> Actuals [ctor assoc format(! o)] .
24     ops CNS REF CAL BLK FRE : −> Control [ctor] .
25
26     op abs : Blk −> Abs [ctor] .
27     op abs : Formals Blk −> Abs [ctor] .
28     op locs : −> LocsAttrib [ctor] .
29     op _:_ : LocsAttrib Set{Loc} −> SemComp [ctor format(c! b! o o)] .
30     …
31  endm
```

Listing 6. Signature for π-lib declarations in Maude

Output in π-lib is produced by command print that side-effects the result of a given expression into the output semantic component.

```
1  mod OUT is
2      ex DEC .
3
4      sort OutAttrib .
5
6      op out : −> OutAttrib [ctor] .
7      op _:_ : OutAttrib ValueStack −> SemComp [ctor format(c! b! o o)] .
8      op print : Exp −> Cmd [ctor format(! o)] .
9      op PRINT : −> Control [ctor] .
10     …
11  endm
```

Listing 7. Signature for π-lib output in Maude

The careful reader most likely noticed that composition of commands was not declared in the CMD module. We declare it in module COMMAND-SEQ-AND-ABNORMAL-TERMINATION together with the exit commands that abruptly terminates the execution of a program. The point is that the semantics of the two constructs (seq and exit) are closely coupled: when exit executes the sequential execution must be interrupted. This is accomplished by means of the exc semantic component that logs when an exit command was executed.

```
1  mod COMMAND−SEQ−AND−ABNORMAL−TERMINATION is
2      ex OUT .
3
4      sorts ExcAttrib Exc .
5
6      op seq : Cmd Cmd −> Cmd [format(! o)] .
7      op exit : Exp −> Cmd [format(! o)] .
8      ops CNT EXT : −> Exc .
9      op EXIT : −> Control .
10     op exc : −> ExcAttrib .
11     op _:_ : ExcAttrib Exc −> SemComp [format(c! b! o o)] .
12     …
13 endm
```

Listing 8. Signature for π-lib abnormal termination in Maude

4.2 π-automata transitions for π-lib dynamic semantics in Maude

Again, due to space constraints, the transition relation is not discussed for the complete π-lib signature. Transitions for identifier and loop evaluation have been chosen to illustrate π-automata transitions for π-lib.

From the EXP module, equations for Identifier and sum evaluation are described. An identifier can represent either a variable or a constant in π-lib. In the former case, given an identifier I, as a result of its declaration, it will be bound to a location l in the environment and l will be mapped to a value, a Rational number or a Boolean value, in the memory store. The evaluation of such an identifier, that is, when I is the top of the control stack, is evaluated by an equation, due to its functional character, by a non-linear pattern together with associative-commutative signature of the semantic components, that guarantees that the same location will appear both in the binding of I in the environment and at the memory store as an index.

```
1      …
2      eq [variable−exp] :
3         < env : (I:Id |−> bind(L:Loc), E:Env),
4           sto : (L:Loc |−> store(R:Rat), S:Store),
5           cnt : (I:Id C:ControlStack), val : SK:ValueStack, ... >
6       =
7         < env : (I:Id |−> bind(L:Loc), E:Env),
8           sto : (L:Loc |−> store(R:Rat), S:Store),
9           cnt : C:ControlStack ,
10          val : (val(R:Rat) SK:ValueStack) , ... > [variant] .
11     …
```

Listing 9. π-automata equations for variable evaluation

The semantics of the loop construction in module CMD is implemented in terms of equations and a rule in Maude. The first equation (i) pushes the loop body into the control stack, (ii) pushes the loop test into the control stack and pushes the whole loop into the value stack. These steps are of functional character, that is, they are Church-Rosser and terminating therefore satisfying the requirements to be implemented by an equation in Maude. The execution of the body of the loop, however, may not terminate as there could be a nested loop, for instance, that does not terminate its execution. For that reason it is implemented as a rule in Maude.

14

```
1     …
2     eq [loop] :
3         < cnt : loop(E:Exp, K:Cmd) C, val : V, ... >
4      =
5         < cnt : E:Exp LOOP C,
6           val : val(loop(E:Exp, K:Cmd)) V, ... > [variant] .
7
8     rl [loop] :
9         < cnt : LOOP C,
10          val : val(true) val(loop(E:Exp, K:Cmd)) V, ... >
11    =>
12        < cnt : K:Cmd loop(E:Exp, K:Cmd) C, val : V, ... > [narrowing] .
13
14    eq [loop] :
15        < cnt : LOOP C,
16          val : val(false) val(loop(E:Exp, K:Cmd)) V, ... >
17     =
18        < cnt : C, val : V, ... > [variant] .
19    …
```

Listing 10. π-automaton rules for loop

Blocks are also implemented in module CMD. A block is a pair composed by a set of declarations and a set of commands or simply a set of commands. When a block is found at the top of the control stack then its declarations and commands are pushed in appropriate order into the control stack. Also, the current set of locations (references to memory cells) in locs semantic component is saved into the value stack together with the current environment. This is done so that memory allocated during the evaluation of a block can be safely freed and the environment recovered after the block evaluation terminates. This behavior is implemented in Maude as equations, due to their functional character. Equation blk-1 is responsible for saving the context and updating control. Equation blk-2 is responsible for recovering control upon block evaluation termination.

```
1     …
2     eq [blk−1] :
3         < cnt : blk(D:Dec, M:Cmd) C, env : E , val : V , locs : SL:Set{Loc} , ... >
4      =
5         < cnt : D:Dec M:Cmd BLK C, env : E ,
6           val : val(E) val(SL:Set{Loc}) V ,
7           locs : noLocs , ... > [variant] .
8
9     eq [blk−2] :
10        < cnt : BLK C ,
11          env : E' ,
12          val : val(E) val(SL:Set{Loc}) V ,
13          locs : SL':Set{Loc},
14          sto : S:Store, ... >
15     =
16        < cnt : C ,
17          env : E ,
18          val : V ,
19          locs : SL:Set{Loc},
20          sto : free(SL':Set{Loc}, S:Store), ... > [variant] .
21    …
```

Listing 11. π-automaton rules for block

Listing 12 presents rules for function call. Operation declarations create a new environment where an identifier is bound to an abstraction, constructed with abs operator, a pair of formal parameters together with a block. An operation call simply

15

pushes into the control stack the identifier representing the operation, the actual parameters and the CAL keyword. This is specified by equation cal-1. Recursion takes care of applying equation prc-id correctly which then pushes the abstraction of the operation being called to value stack. Finally, when control keyword CAL is found on the top of the control stack and an abstraction is found in the value stack with its (fully evaluated by equation act) actual arguments on top of it, equation cal-2 produces a block with the body of the abstraction as command and bindings between formals and actuals as declarations. (In functional programming jargon, this semantics follows the idea of reducing applications to let expressions.)

```
1    …
2    eq [cal−1] :
3        < cnt : cal(I:Id, A:Actuals) C, ... > =
4        < cnt : I:Id A:Actuals CAL C, ... > [variant] .
5
6    eq [cal−2] :
7        < cnt : CAL C,
8          val : V1:ValueStack
9          val(abs(F:Formals, B:Blk)) V2:ValueStack, ... > =
10       < cnt : addDec(match(F:Formals, V1:ValueStack), B:Blk) C,
11         val : V2:ValueStack, ... > [variant] .
12
13   eq [prc−id] :
14       < cnt : (I:Id C),
15         env : (I:Id |−> A:Abs, E),
16         val : V , ... > =
17       < cnt : C,
18         env : (I:Id |−> A:Abs, E),
19         val : (val(A:Abs) V), ... > [variant] .
20
21   eq [act] :
22       < cnt : act(E:Exp, A:Actuals) C, ... > =
23       < cnt : A:Actuals E:Exp C, ... > [variant] .
24   …
```

Listing 12. π-automaton rules for function call

4.3 IMP: execution, invariant checking and model checking

In this Section, the use of π-lib is illustrated by a compiler for a simple (and yet Turing-complete) imperative language called IMP. Once an implementation of π-lib is in place, such as the Maude implementation discussed in Sections 4.1 and 4.2, one has to write: (i) a parser for IMP and (ii) a transformer from IMP AST to π-lib. Everything else is handled by π-lib-based tools such as interpretation, code-generation and validation. The current implementation of π-lib in Maude supports execution by rewriting, symbolic execution by narrowing and LTL model-checking. These are the tools that are "lifted" from Maude to IMP.

The big picture A Maude compiler for a language L, such as IMP, defined as a denotation of π-lib constructions, has the following main components: (i) A read-eval-loop function (or command-line interface) that invokes different meta-functions depending on the given command. For example, a load command invokes the parser, exec invokes metaRewrite and mc invokes metaReduce with the model checker. (ii) A parser for L, which is essentially a meta-function that given a list of qids returns

16

a meta-term according to a given grammar, specified as a functional module; (iii) A transformer from L to π-lib, a meta-function that given a term in the data-type of the source language, produces a term on the data-type of the target languge; (iv) A pretty-printer from π-lib to L, a meta-function that given a term in the data-type of the target language produces a list of qids.

Figure 5 illustrates the execution of model checking an IMP program, implementing a Mutex protocol, for safety and liveness properties. State propositions p_1 and p_2 are *automatically* generated by the compiler to properly construct the π-automata for Mutex, as discussed in Section 3.4.

IMP*'s command-line interface* In Listing 13 we describe an excerpt of the implementation for IMP's command-line interface, detailing only the module inclusions, sort declarations for the command-line state and rules for loading an IMP program. A full description is not possible due to space constraints. However, the pattern explained in this excerpt is the same for every command: a qidlist denotes the input, a different meta-function is called on them, depending on the input, that updates or not the state of IMP's command-line interface and or the output of the system as whole, with a message to the end user.

First, module IMP-INTERFACE includes LOOP-MODE module. Module LOOP-MODE declares operation op [_,_,_] : QidList State QidList -> System that declares the state of the loop-mode instance, in our case, IMP loop mode. IMP-INTERFACE than renames LOOP-MODE' sort State to LoopState, to avoid a name clash. Next, modules COMPILE-IMP-TO-π-lib and IMP-PRETTY-PRINTING are included, responsible for the functionalities that their names stand for. Operation op <_;_;_> : MetaIMPModule Dec? QidList -> IMPState represents the state of the command-line interface, which is a triple comprised by (i) the meta-representation of an IMP module, (ii) the π-lib representation of the IMP module in the first projection, and (iii) a qid list denoting the message from the last processed command. Rule labeled in is responsible for processing the input of an IMP module, therefore, in this case, the first projection of the System term contains a list of qids representing an IMP module. Should the parsing process be successful, variable T:ResultPair? will be bound to a pair whose first projection is the term resulting from parsing and in the second projection its sort, or a unary function, of sort ResultPair?, denoting that the parsing process was not properly carried on, with a parameter representing the qid where the parsing process failed. With a successful parsing, the following term becomes now the state of the system

```
1  < getTerm(T:ResultPair?) ; compileMod(getTerm(T:ResultPair?)) ;
2      'IMP: '\b 'Module Q:Qid 'loaded. '\o >
```

where getTerm(T:ResultPair?) denotes the meta-term, according to IMP's grammar, representing the input module, compileMod(getTerm(T:ResultPair?)) denotes the input module in π-lib, in the second projection, and the third component of the IMPState triple is a qidlist represents a message to the user know informing that the module was properly loaded.

```
1  mod IMP−INTERFACE is
2    pr LOOP−MODE ∗ (sort State to LoopState).
```

```
                            1. bash

IMP Prototype (March 2018)
IMP: Module Mutex loaded.
module Mutex
    var p1, p2
    const idle, wait, crit
    init p1 = 0, p2 = 0, idle = 0, wait = 1, crit = 2
    proc mutex {
        while (true) do {
            if (p1 == idle \/ p2 == idle) {
                p1 := wait | p2 := wait
            } else if (p1 == idle \/ p2 == wait) {
                p1 := wait | p2 := crit
            } else if (p1 == idle \/ p2 == crit) {
                p1 := wait | p2 := idle
            } else if (p1 == wait \/ p2 == idle) {
                p1 := crit | p2 := wait
            } else if (p1 == wait \/ p2 == wait) {
                p1 := crit | p2 := crit
            } else if (p1 == wait \/ p2 == crit) {
                p2 := idle
            } else if (p1 == crit \/ p2 == idle) {
                p1 := idle | p2 := wait
            } else if (p1 == crit \/ p2 == wait) {
                p1 := idle
            } else nop
        }
    }
end
rewrites: 841 in 35ms cpu (36ms real) (23605 rewrites/second)
IMP: Model check result to command mutex() |= []~(p1(2)/\ p2(2)):
true
rewrites: 768 in 16ms cpu (16ms real) (47542 rewrites/second)
IMP: Model check counter example to command mutex() |= [](p1(1)->
    <> p1(2)):
Path from the initial state:
while(true)...[p1 = 0 p2 = 0]-> while(true)...[p1 = 0 p2 = 0]-> p2
    := wait | p1 := wait[p1 = 0 p2 = 0]
Loop:
while(true)...[p1 = 1 p2 = 0]-> p2 := wait | p1 := crit[p1 = 1 p2 =
    0]-> while(true)...[p1 = 1 p2 = 1]-> p2 := crit | p1 := crit[p1
    = 1 p2 = 1]-> while(true)...[p1 = 1 p2 = 2]
```

Fig. 5. Model checking a Mutex protocol in IMP that is safe but not live.

```
3    pr COMPILE−IMP−TO−π−lib .
4    pr IMP−PRETTY−PRINTING .
5
6    sorts Dec? MetaIMPModule Command IMPState .
7    subsort Term < MetaIMPModule .
8    subsort Dec < Dec? .
9    subsort IMPState < LoopState .
10
11   op noDec : −> Dec? .
12   op noModule : −> MetaIMPModule .
13   op idle : −> Command .
14   op <_;_;_> : MetaIMPModule Dec? QidList −> IMPState .
15
16   …
17   vars QIL QIL' QIL'' QIL1 QIL2 : QidList .
18
19   ∗∗∗ Loading a module.
20   crl [in] : ['module Q:Qid QIL, < M:MetaIMPModule ; D:Dec? ; QIL' >, QIL''] =>
21              if (T:ResultPair? :: ResultPair)
22              then
23                  [nil, < getTerm(T:ResultPair?) ;
24                          compileMod(getTerm(T:ResultPair?)) ;
25                          'IMP: '\b 'Module Q:Qid 'loaded. '\o >, QIL'']
26              else [nil, < noModule ; noDec ; nil >,
27                      printParseError('module Q:Qid QIL, T:ResultPair?)]
28              fi
29   if T:ResultPair? :=
30      metaParse(upModule('IMP−GRAMMAR, false),
31        'module Q:Qid QIL, 'ModuleDecl) .
32
33   ∗∗∗ Viewing a module.
34   crl [view] : ['view , < M:MetaIMPModule ; D:Dec ; QIL' >, QIL''] =>
35                  [nil, < M:MetaIMPModule ; D:Dec ; QIL' >,
36                          printModule(M:MetaIMPModule)]
37   if M:MetaIMPModule =/= noModule .
38
39   …
40   endm
```

Listing 13. IMP's command-line interface in Maude

IMP *parser* To write a parser in Maude one has to first define the grammar of the language as a Maude functional module. The IMP-GRAMMAR module is the first argument in the function call to metaParse in Rule `in` of module IMP-INTERFACE in Listing 13. Essentially, variables (or non-terminals) are represented by sorts, terminals by constants and grammar rules are represented by operations. Listing 14 encodes an excerpt of the IMP grammar, only enough to discuss the main elements of the grammar representation: we exemplify it with sum expressions Sort Expression-Decl, for instance, specifies both arithmetic and Boolean expressions. A grammar rule relating two grammar variables is represented by a subsort declaration. Therefore, PredicateDecl, the sort for Boolean expressions, is a subsort of ExpressionDecl.

```
1    fmod IMP−GRAMMAR is
2     pr TOKEN . pr RAT .
3     inc PREDICATE−DECL . inc COMMAND−DECL .
4     sorts VariablesDecl ConstantsDecl OperationsDecl ProcDeclList
5          ProcDecl FormalsDecl BlockCommandDecl ExpressionDecl
6          InitDecl InitDeclList InitDecls ClausesDecl
7          ModuleDecl Expression .
8
9     subsort InitDecl < InitDeclList .
10    subsort VariablesDecl ConstantsDecl ProcDeclList InitDecls < ClausesDecl .
```

19

```
11   subsort BlockCommandDecl < CommandDecl .
12   subsort ProcDecl < ProcDeclList .
13   subsort PredicateDecl < ExpressionDecl .
14
15   . . .
16
17   *** Arithmetic expressions
18   op _+_ : Token Token -> ExpressionDecl [gather(e E) prec 15] .
19   op _+_ : Token ExpressionDecl -> ExpressionDecl [gather(e E) prec 15] .
20   op _+_ : ExpressionDecl Token -> ExpressionDecl [gather(e E) prec 15] .
21   op _+_ : ExpressionDecl ExpressionDecl -> ExpressionDecl [gather(e E) prec 15] .
22
23   . . .
24   endfm
```

Listing 14. IMP grammar as a Maude functional module

The `prec` operator attribute declares the precedence of the operator, as commonly available in parser generators. A precedence value associates an Integer greater than 0 to terms with the given operator at the top. The gather attribute allows for the specification of precedence constraints among the operands and an operator. For instance, the pattern `gather(e, E)` specifies that the first operand must have strictly lower precedence than the given operator, while the second operand must have a lower or equal precedence with respect to the given operator.

IMP *to π-lib transformer* Compilation from IMP to π-lib is quite trivial as there exists a one-to-one correspondence between IMP constructions and π-lib. Essentially, an IMP `module` gives rise to a π-lib `dec`. IMP `var` and `const` are declarations and so is a `proc` declaration that gives rise to a `prc` declaration in π-lib.

```
1   mod COMPILE-IMP-TO-π-lib is
2     inc PREDICATE-DECL .
3     inc COMMAND-DECL .
4     pr π-lib .
5     pr META-LEVEL .
6
7   . . .
```

The compilation from IMP to π-lib exp relates IMP tokens to π-lib Id, IMP arithmetic and boolean expressions to π-lib Exp. In particular, the compilation of a `token` has to check if the token is a primitive type, either `Rat` (for Rational numbers) or `Bool` (for Boolean values), or an identifier. Since `Rat` and `Bool` are tokenized and we need Maude metalevel descent function `downTerm` to help us parse them into proper constants

```
1   op compileId : Qid -> Id .
2   eq compileId(I:Qid) = idn(downTerm(I:Qid, 'Qid)) .
3
4   op compileId : Term -> Id .
5   eq compileId('token[I:Qid]) =
6       if (metaParse(upModule('RAT, false),
7            downTerm(I:Qid, 'Qid), 'Rat) :: ResultPair)
8         then rat(downTerm(getTerm(metaParse(upModule('RAT, false),
9              downTerm(I:Qid, 'Qid), 'Rat)), 1/2))
10      else
11        if (metaParse(upModule('BOOL, false),
12           downTerm(I:Qid, 'Qid), 'Bool) :: ResultPair)
13        then boo(downTerm(getTerm(metaParse(upModule('BOOL, false),
14                 downTerm(I:Qid, 'Qid), 'Bool)), true))
```

```
15      else idn(downTerm(I:Qid, 'Qid))
16    fi
17  fi .
```

Arithmetic expressions are mapped to their prefixed counterpart in π-lib, e.g, an IMP expression `a + b` is compiled to `add(compileExp(a), compileExp(b))`.

```
1  op compileExp : Term −> Exp .
2  ceq compileExp(I:Qid) = compileId(I:Qid)
3   if not(I:Qid :: Constant) .
4  eq compileExp('token[I:Qid]) = compileId('token[I:Qid]) .
5  eq compileExp('_+_[T1:Term, T2:Term]) =
6      add(compileExp(T1:Term), compileExp(T2:Term)) .
7  eq compileExp('_−_[T1:Term, T2:Term]) =
8      sub(compileExp(T1:Term), compileExp(T2:Term)) .
9  eq compileExp('_*_[T1:Term, T2:Term]) =
10     mul(compileExp(T1:Term), compileExp(T2:Term)) .
11 eq compileExp('_/_[T1:Term, T2:Term]) =
12     div(compileExp(T1:Term), compileExp(T2:Term)) .
```

5   The complete π-lib code in Maude

$\langle gsmc \rangle \equiv$

```
  fmod VALUE-SORT is
      sort Value .
  endfm

  fmod CONTROL-SORT is
      sort Control .
  endfm

  view Control from TRIV to CONTROL-SORT is
      sort Elt to Control .
  endv

  view Value from TRIV to VALUE-SORT is
      sort Elt to Value .
  endv

  -- Note: AI theories are not currently supported by Maude unification,
  -- as of Alpha 115.
  fmod GNELIST{X :: TRIV} is
      pr NAT .
      sorts GNeList{X} .
      subsort X$Elt < GNeList{X} .

      op __ : GNeList{X} GNeList{X} -> GNeList{X} [ctor assoc prec 25] .
  endfm

  -------------------------------
```

```
-- GSMC semantics for basic programming language constructs .

fmod VALUE-STACK is
    pr GNELIST{Value} * (sort GNeList{Value} to NeValueStack) .
    sort ValueStack .
    subsort NeValueStack < ValueStack .
    op evs : -> ValueStack .
    op __ : ValueStack ValueStack -> ValueStack [ditto] .
endfm

fmod CONTROL-STACK is
    pr GNELIST{Control} * (sort GNeList{Control} to NeControlStack) .
    sort ControlStack .
    subsort NeControlStack < ControlStack .
    op ecs : -> ControlStack .
    op __ : ControlStack ControlStack -> ControlStack [ditto] .
endfm

fmod MEMORY-SORTS is
    sorts Loc Storable .
endfm

view Loc from TRIV to MEMORY-SORTS is
    sort Elt to Loc .
endv

view Storable from TRIV to MEMORY-SORTS is
    sort Elt to Storable .
endv

fmod MEMORY is
    pr NAT .
    ex MAP{Loc,Storable} * (sort Entry{Loc,Storable} to Cell,
        sort Map{Loc,Storable} to Store,
        op undefined to undefloc, op empty to noStore).
    ex SET{Loc} * (op empty to noLocs) .

    op loc : Nat -> Loc [ctor] .
    op newLoc : Store -> Loc .
    op $newLoc : Store Nat -> Loc .
    eq newLoc(noStore) = loc(0) .
    ceq newLoc(S:Store) = $newLoc(S:Store, 0) if S:Store =/= noStore .
    eq $newLoc(noStore, N:Nat) = loc(N:Nat + 1) .
    ceq $newLoc((S:Store, loc(N:Nat) |-> O:Storable), N':Nat) =
        $newLoc(S:Store, N:Nat) if N:Nat >= N':Nat .
```

```
    ceq $newLoc((S:Store, loc(N:Nat) |-> O:Storable), N':Nat) =
        $newLoc(S:Store, N':Nat) if N:Nat < N':Nat .

    op $free : Loc Store -> Store .
    eq $free(L:Loc, ((L:Loc |-> O:Storable), S:Store)) = S:Store .
    eq $free(L:Loc, S:Store) = S:Store [owise] .

    op free : Set{Loc} Store -> Store .
    eq free(noLocs, S:Store) = S:Store .
    eq free((L:Loc , SL:Set{Loc}), S:Store) =
        free(SL:Set{Loc}, $free(L:Loc, S:Store)) [owise] .
endfm

fmod ENV-SORTS is
    sorts Id Bindable .
endfm

view Id from TRIV to ENV-SORTS is sort Elt to Id . endv

view Bindable from TRIV to ENV-SORTS is sort Elt to Bindable . endv

fmod ENV is
    ex MAP{Id,Bindable} * (sort Entry{Id,Bindable} to Bind,
        sort Map{Id,Bindable} to Env,
        op undefined to undefid, op empty to noEnv).
endfm

--------------
-- Generalized SMC machines.

fmod SEMANTIC-COMPONENTS is
    sorts SemComp .
endfm

view SemComp from TRIV to SEMANTIC-COMPONENTS is
    sort Elt to SemComp .
endv

fmod GSMC is
    ex VALUE-STACK .
    ex MEMORY .
    ex CONTROL-STACK .
    ex ENV .
    ex SET{SemComp} * (op empty to noSemComp) .
```

```
        sorts Attrib Conf EnvAttrib StoreAttrib ControlAttrib ValueAttrib .
        subsort EnvAttrib StoreAttrib ControlAttrib ValueAttrib < Attrib   .
        op <_> : Set{SemComp} -> Conf [format(c! c! c! o)] .

        -- Semantic components
        op env : -> EnvAttrib .
        op sto : -> StoreAttrib .
        op cnt : -> ControlAttrib .
        op val : -> ValueAttrib .
        op _:_ : EnvAttrib Env -> SemComp [ctor format(c! b! o o)] .
        op _:_ : StoreAttrib Store -> SemComp [ctor format(r! b! o o)] .
        op _:_ : ControlAttrib ControlStack -> SemComp [ctor format(c! b! o o)] .
        op _:_ : ValueAttrib ValueStack -> SemComp [ctor format(c! b! o o)] .
endfm
```

6 Basic Programming Languages Constructs

– Expressions: identifier, rational numbers, Boolean values, arithmetic and predicates. Semantic components: Identifiers require the environment and the store. The remaining constructs do not require any semantic component.
– Commands: print, assignment, sequence, choice, block, conditional, and loop. Print updates output. The remaining constructs affect only the store.
– Declarations: variables, constants, and procedures. These constructs affect the environment.
– Abnormal termination: exit. Command exit changes the semantic component that indicates abnormal termination. The behavior of command composition is also affected by this semantic component.

6.1 Expressions

⟨*bplc-exp*⟩≡

```
fmod EXP is
    pr QID .
    pr RAT .
    pr GSMC .

    sorts Exp BExp AExp .
    subsort Id < BExp AExp < Exp < Control .

    -- Identifiers
    op idn : Qid -> Id [ctor format(!g o)] .

    -- Arithmetic
    op rat : Rat -> AExp [ctor format(!g o)] .
    op add : AExp AExp -> AExp [format(! o)] .
    op sub : AExp AExp -> AExp [format(! o)] .
    op mul : AExp AExp -> AExp [format(! o)] .
    op div : AExp AExp -> AExp [format(! o)] .

    ops ADD SUB MUL DIV : -> Control [ctor] .

    -- Boolean expressions
    op boo : Bool -> AExp [ctor format(!g o)] .
    op gt : Exp Exp -> BExp [format(! o)] .
    op ge : Exp Exp -> BExp [format(! o)] .
    op lt : Exp Exp -> BExp [format(! o)] .
    op le : Exp Exp -> BExp [format(! o)] .
    op eq : Exp Exp -> BExp [format(! o)] .
    op neg : BExp -> BExp [format(! o)] .
    op and : BExp BExp -> BExp [format(! o)] .
    op or : BExp BExp -> BExp [format(! o)] .
```

25

```
ops LT LE EQ NEG AND OR : -> Control [ctor] .

op store : Rat -> Storable [ctor format(ru! o)] .
op store : Bool -> Storable [ctor format(ru! o)] .
op bind : Loc -> Bindable [ctor] .
op bind : Rat -> Bindable [ctor] .
op bind : Bool -> Bindable [ctor] .

op val : Storable -> Value [ctor] .
op val : Rat -> Value [ctor] .
op val : Bool -> Value [ctor] .
op val : Loc -> Value [ctor] .
op val : Id -> Value [ctor] .

var ... : Set{SemComp} .

eq gt(E1:Exp, E2:Exp) = neg(le(E1:Exp, E2:Exp)) .
eq ge(E1:Exp, E2:Exp) = neg(lt(E1:Exp, E2:Exp)) .

eq [rat-exp] :
    < cnt : (rat(R:Rat) C:ControlStack), val : SK:ValueStack, ... >
 =
    < cnt : C:ControlStack,
      val : (val(R:Rat) SK:ValueStack), ... > [variant] .

eq [bool-exp] :
    < cnt : (boo(B:Bool) C:ControlStack), val : SK:ValueStack, ... >
 =
    < cnt : C:ControlStack,
      val : (val(B:Bool) SK:ValueStack), ... > [variant] .

eq [add-exp] :
    < cnt : (add(E1:Exp, E2:Exp) C:ControlStack), ... >
 =
    < cnt : (E1:Exp E2:Exp ADD C:ControlStack), ... > [variant] .

eq [add-exp] :
    < cnt : (ADD C:ControlStack),
      val : (val(R1:Rat) val(R2:Rat) SK:ValueStack), ... >
 =
    < cnt : C:ControlStack,
      val : (val(R1:Rat + R2:Rat) SK:ValueStack), ... > [variant] .

eq [sub-exp] :
```

```
   < cnt : (sub(E1:Exp, E2:Exp) C:ControlStack), ... >
=
   < cnt : (E1:Exp E2:Exp SUB C:ControlStack), ... > [variant] .

eq [sub-exp] :
   < cnt : (SUB C:ControlStack),
     val : (val(R1:Rat) val(R2:Rat) SK:ValueStack), ... >
=
   < cnt : C:ControlStack,
     val : (val(R2:Rat - R1:Rat) SK:ValueStack), ... > [variant] .

eq [mul-exp] :
   < cnt : (mul(E1:Exp, E2:Exp) C:ControlStack), ... >
=
   < cnt : (E1:Exp E2:Exp MUL C:ControlStack), ... > [variant] .

eq [mul-exp] :
   < cnt : (MUL C:ControlStack),
     val : (val(R1:Rat) val(R2:Rat) SK:ValueStack), ... >
=
   < cnt : C:ControlStack,
     val : (val(R1:Rat * R2:Rat) SK:ValueStack), ... > [variant] .

eq [div-exp] :
   < cnt : (div(E1:Exp, E2:Exp) C:ControlStack), ... >
=
   < cnt : (E1:Exp E2:Exp DIV C:ControlStack), ... > [variant] .

eq [div-exp] :
   < cnt : (DIV C:ControlStack),
     val : (val(R1:Rat) val(R2:NzRat) SK:ValueStack), ... >
=
   < cnt : C:ControlStack,
     val : (val(R1:Rat / R2:NzRat) SK:ValueStack), ... > [variant] .

eq [lt-exp] :
   < cnt : (lt(E1:Exp, E2:Exp) C:ControlStack), ... >
=
   < cnt : (E2:Exp E1:Exp LT C:ControlStack), ... > [variant] .

eq [lt-exp] :
   < cnt : (LT C:ControlStack),
     val : (val(R1:Rat) val(R2:Rat) SK:ValueStack), ... >
=
   < cnt : C:ControlStack,
```

```
            val : (val(R1:Rat < R2:Rat) SK:ValueStack), ... > [variant] .

eq [le-exp] :
    < cnt : (le(E1:Exp, E2:Exp) C:ControlStack), ... >
 =
    < cnt : (E2:Exp E1:Exp LE C:ControlStack), ... > [variant] .

eq [le-exp] :
    < cnt : (LE C:ControlStack),
      val : (val(R1:Rat) val(R2:Rat) SK:ValueStack), ... >
 =
    < cnt : C:ControlStack,
      val : (val(R1:Rat <= R2:Rat) SK:ValueStack), ... > [variant] .

eq [eq-exp] :
    < cnt : (eq(E1:Exp, E2:Exp) C:ControlStack), ... >
 =
    < cnt : (E2:Exp E1:Exp EQ C:ControlStack), ... > [variant] .

eq [eq-exp] :
    < cnt : (EQ C:ControlStack),
      val : (val(R1:Rat) val(R2:Rat) SK:ValueStack), ... >
 =
    < cnt : C:ControlStack,
      val : (val(R1:Rat == R2:Rat) SK:ValueStack), ... > [variant] .

eq [eq-exp] :
    < cnt : (EQ C:ControlStack),
      val : (val(B1:Bool) val(B2:Bool) SK:ValueStack), ... >
 =
    < cnt : C:ControlStack,
      val : (val(B1:Bool == B2:Bool) SK:ValueStack), ... > [variant] .

eq [neg-exp] :
    < cnt : (neg(E:Exp) C:ControlStack), ... >
 =
    < cnt : (E:Exp NEG C:ControlStack), ... > [variant] .

eq [neg-exp] :
    < cnt : (NEG C:ControlStack),
      val : (val(B:Bool) SK:ValueStack), ... >
 =
    < cnt : C:ControlStack,
      val : (val(not(B:Bool)) SK:ValueStack), ... > [variant] .
```

```
eq [and-exp] :
    < cnt : (and(E1:Exp, E2:Exp) C:ControlStack), ... >
 =
    < cnt : (E1:Exp E2:Exp AND C:ControlStack), ... > [variant] .

eq [and-exp] :
    < cnt : (AND C:ControlStack),
      val : (val(B1:Bool) val(B2:Bool) SK:ValueStack), ... >
 =
    < cnt : C:ControlStack,
      val : (val(B1:Bool and B2:Bool) SK:ValueStack), ... > [variant] .

eq [or-exp] :
    < cnt : (or(E1:Exp, E2:Exp) C:ControlStack), ... >
 =
    < cnt : (E1:Exp E2:Exp OR C:ControlStack), ... > [variant] .

eq [and-exp] :
    < cnt : (OR C:ControlStack),
      val : (val(B1:Bool) val(B2:Bool) SK:ValueStack), ... >
 =
    < cnt : C:ControlStack,
      val : (val(B1:Bool or B2:Bool) SK:ValueStack), ... > [variant] .

eq [variable-exp] :
    < env : (I:Id |-> bind(L:Loc), E:Env),
      sto : (L:Loc |-> store(R:Rat), S:Store),
      cnt : (I:Id C:ControlStack), val : SK:ValueStack, ... >
 =
    < env : (I:Id |-> bind(L:Loc), E:Env),
      sto : (L:Loc |-> store(R:Rat), S:Store),
      cnt : C:ControlStack ,
      val : (val(R:Rat) SK:ValueStack) , ... > [variant] .

eq [variable-exp] :
    < env : (I:Id |-> bind(L:Loc), E:Env),
      sto : (L:Loc |-> store(B:Bool), S:Store),
      cnt : (I:Id C:ControlStack), val : SK:ValueStack, ... >
 =
    < env : (I:Id |-> bind(L:Loc), E:Env),
      sto : (L:Loc |-> store(B:Bool), S:Store),
      cnt : C:ControlStack ,
      val : (val(B:Bool) SK:ValueStack) , ... > [variant] .

eq [constant-rat-exp] :
```

```
          < env : (I:Id |-> bind(R:Rat), E:Env), cnt : (I:Id C:ControlStack),
            val : SK:ValueStack , ... >
      =
          < env : (I:Id |-> bind(R:Rat), E:Env), cnt : C:ControlStack,
            val : (val(R:Rat) SK:ValueStack), ... > [variant] .

   eq [constant-bool-exp] :
          < env : (I:Id |-> bind(B:Bool), E:Env), cnt : (I:Id C:ControlStack),
            val : SK:ValueStack, ... >
      =
          < env : (I:Id |-> bind(B:Bool), E:Env), cnt : C:ControlStack,
            val : (val(B:Bool) SK:ValueStack), ... > [variant] .
endfm
```

6.2 Commands

⟨*bplc-cmd*⟩≡
```
  mod CMD is
      pr EXP .

      sort Cmd .
      subsort Cmd < Control .

      op nop : -> Cmd [ctor format(! o)] .
      op choice : Cmd Cmd -> Cmd [ctor assoc comm format(! o)] .
      op assign : Id Exp -> Cmd [ctor format(! o)] .
      op loop : Exp Cmd -> Cmd [ctor format(! o)] .
      op if : Exp Cmd Cmd -> Cmd [ctor format(! o)] .

      ops ASSIGN LOOP IF : -> Control [ctor] .

      op val : Cmd -> Value [ctor] .

      var ... : Set{SemComp} . var E : Env . var S : Store .
      var C : ControlStack . var V : ValueStack .

      eq [nop-cmd] :
          < cnt : nop C, ... > = < cnt : C, ... > [variant] .

      rl [choice-cmd] :
          < cnt : choice(M1:Cmd, M2:Cmd) C, ... > =>
          < cnt : M1:Cmd C, ... > . -- [narrowing] .

      eq [assign-cmd] :
          < env : (I:Id |-> bind(L:Loc), E),
            cnt : (assign(I:Id, E:Exp) C),
            val : V, ... >
       =
          < env : (I:Id |-> bind(L:Loc), E),
            cnt : (E:Exp ASSIGN C),
            val : (val(I:Id) V), ... > [variant] .

      eq [assign-cmd] :
          < env : (I:Id |-> bind(L:Loc), E),
            sto : (L:Loc |-> T:Storable, S),
            cnt : (ASSIGN C),
            val : (val(R:Rat) val(I:Id) V), ... >
       =
          < env : (I:Id |-> bind(L:Loc), E),
            sto : (L:Loc |-> store(R:Rat), S),
```

31

```
              cnt : C,
              val : V, ... > [variant] .

    eq [assign-cmd] :
        < env : (I:Id |-> bind(L:Loc), E),
          sto : (L:Loc |-> T:Storable, S),
          cnt : (ASSIGN C),
          val : (val(B:Bool) val(I:Id) V), ... >
     =
        < env : (I:Id |-> bind(L:Loc), E),
          sto : (L:Loc |-> store(B:Bool), S),
          cnt : C,
          val : V, ... > [variant] .

    eq [loop] :
        < cnt : loop(E:Exp, K:Cmd) C, val : V, ... >
     =
        < cnt : E:Exp LOOP C,
          val : val(loop(E:Exp, K:Cmd)) V, ... > [variant] .

    rl [loop] :
        < cnt : LOOP C,
          val : val(true) val(loop(E:Exp, K:Cmd)) V, ... >
    =>
        < cnt : K:Cmd loop(E:Exp, K:Cmd) C,
          val : V, ... > . -- [narrowing] .

    eq [loop] :
        < cnt : LOOP C,
          val : val(false) val(loop(E:Exp, K:Cmd)) V, ... >
     =
        < cnt : C, val : V, ... > [variant] .

    eq [if] :
      < cnt : if(E:Exp, K1:Cmd, K2:Cmd) C, val : V, ... >
     =
      < cnt : E:Exp IF C,
        val : val(if(E:Exp, K1:Cmd, K2:Cmd)) V, ... > [variant] .

    eq [if] :
      < cnt : IF C,
        val : val(true) val(if(E:Exp, K1:Cmd, K2:Cmd)) V, ... >
     =
      < cnt : K1:Cmd C,
        val : V, ... > [variant] .
```

32

```
    eq [if] :
      < cnt : IF C,
        val : val(false) val(if(E:Exp, K1:Cmd, K2:Cmd)) V, ... >
     =
      < cnt : K2:Cmd C,
        val : V, ... > [variant] .
endm
```

6.3 Declarations

⟨*bplc-dec*⟩≡
```
mod DEC is
    ex CMD .

    sorts Abs Blk Dec Formal Formals Actual Actuals LocsAttrib .
    subsort Actuals Dec < Control .
    subsort Formal < Formals .
    subsort Exp < Actual < Actuals .
    subsort Blk < Cmd .
    subsort Abs < Bindable .

    op cns : Id Exp -> Dec [ctor format(! o)] .
    op ref : Id Exp -> Dec [ctor format(! o)] .
    op prc : Id Blk -> Dec [ctor format(! o)] .
    op prc : Id Formals Blk -> Dec [ctor format(! o)] .
    op par : Id -> Formal [ctor format(! o)] .
    op vod : -> Formal [ctor format(! o)] .
    op for : Formals Formals -> Formals [ctor assoc format(! o)] .
    op dec : Dec Dec -> Dec [ctor format(! o)] .
    op blk : Cmd -> Blk [ctor format(! o)] .
    op blk : Dec Cmd -> Blk [ctor format(! o)] .
    op cal : Id -> Cmd [ctor format(! o)] .
    op cal : Id Actuals -> Cmd [ctor format(! o)] .
    op act : Actuals Actuals -> Actuals [ctor assoc format(! o)] .
    ops CNS REF CAL BLK FRE : -> Control [ctor] .

    op val : Env -> Value [ctor] .
    op val : Loc -> Value [ctor] .
    op val : Abs -> Value [ctor] .
    op val : Set{Loc} -> Value [ctor] .

    op abs : Blk -> Abs [ctor] .
    op abs : Formals Blk -> Abs [ctor] .
    op locs : -> LocsAttrib [ctor] .
    op _:_ : LocsAttrib Set{Loc} -> SemComp [ctor format(c! b! o o)] .

    var ... : Set{SemComp} . vars E E' : Env . var S : Store .
    var C : ControlStack . var V : ValueStack .

    eq [blk] :
        < cnt : blk(D:Dec, M:Cmd) C, env : E , val : V , locs : SL:Set{Loc} , ... >
     =
        < cnt : D:Dec M:Cmd BLK C, env : E ,
                val : val(E) val(SL:Set{Loc}) V ,
```

34

```
                               locs : noLocs , ... > [variant] .

        eq [blk] :
            < cnt : blk(M:Cmd) C, env : E , val : V , locs : SL:Set{Loc} , ... >
         =
            < cnt : M:Cmd BLK C, env : E ,
                    val : val(E) val(SL:Set{Loc}) V ,
                    locs : noLocs , ... > [variant] .

        eq [blk] :
            < cnt : BLK C ,
              env : E' ,
              val : val(E) val(SL:Set{Loc}) V ,
              locs : SL':Set{Loc},
              sto : S:Store, ... >
         =
            < cnt : C ,
              env : E ,
              val : V ,
              locs : SL:Set{Loc},
              sto : free(SL':Set{Loc}, S:Store), ... > [variant] .

        eq [ref] :
            < cnt : ref(I:Id, X:Exp) C , val : V , ... > =
            < cnt : X:Exp REF C , val : val(I:Id) V , ... > [variant] .

        eq [ref] :
            < cnt : REF C, env : E ,
              sto : S ,
              val : val(R:Rat) val(I:Id) V ,
              locs : SL:Set{Loc} , ... >
         =
            < cnt : C ,
              env : insert(I:Id, bind(newLoc(S)), E) ,
              sto : insert(newLoc(S), store(R:Rat), S) ,
              val : V ,
              locs : (newLoc(S) , SL:Set{Loc}) , ... > [variant] .

        eq [cns] :
            < cnt : cns(I:Id, X:Exp) C , val : V , ... > =
            < cnt : X:Exp CNS C , val : val(I:Id) V , ... > [variant] .

        eq [cns] :
            < cnt : CNS C, env : E , val : val(R:Rat) val(I:Id) V , ... > =
            < cnt : C ,
```

```
            env : (I:Id |-> bind(R:Rat) , E) ,
            val : V , ... > [variant] .

    eq [prc] :
        < cnt : prc(I:Id, F:Formals, B:Blk) C, env : E, ... > =
        < cnt : C,
          env : insert(I:Id, abs(F:Formals, B:Blk), E), ... > [variant] .

    eq [prc] :
        < cnt : prc(I:Id, B:Blk) C, env : E, ... > =
        < cnt : C,
          env : insert(I:Id, abs(B:Blk), E), ... > [variant] .

    eq [dec] :
        < cnt : dec(D1:Dec, D2:Dec) C, ... > =
        < cnt : D1:Dec D2:Dec C , ... > [variant] .

    eq [cal] :
        < cnt : cal(I:Id) C, ... > =
        < cnt : I:Id CAL C, ... > [variant] .

    eq [cal] :
        < cnt : cal(I:Id, A:Actuals) C, ... > =
        < cnt : I:Id A:Actuals CAL C, ... > [variant] .

    eq [cal] :
        < cnt : CAL C,
          val : V1:ValueStack
          val(abs(F:Formals, B:Blk)) V2:ValueStack, ... > =
        < cnt : addDec(match(F:Formals, V1:ValueStack), B:Blk) C,
          val : V2:ValueStack, ... > [variant] .

    eq [cal] :
        < cnt : CAL C,
          val : val(abs(B:Blk)) V:ValueStack, ... > =
        < cnt : B:Blk C,
          val : V:ValueStack, ... > [variant] .

    eq [prc-id] :
        < cnt : (I:Id C),
          env : (I:Id |-> A:Abs, E),
          val : V , ... > =
        < cnt : C,
          env : (I:Id |-> A:Abs, E),
          val : (val(A:Abs) V), ... > [variant] .
```

36

```
      eq [act] :
          < cnt : act(E:Exp, A:Actuals) C, ... > =
          < cnt : A:Actuals E:Exp C, ... > [variant] .

      op match : Formals ValueStack -> Dec .
      eq match(par(I:Id), val(R:Rat)) = ref(I:Id, rat(R:Rat)) .
      eq match(par(I:Id), val(B:Bool)) = ref(I:Id, boo(B:Bool)) .
      eq match(for(F:Formal, L:Formals), (V:Value VS:ValueStack)) =
          dec(match(F:Formal, V:Value),
            match(L:Formals, VS:ValueStack)) .

      op addDec : Dec Blk -> Blk .
      eq addDec(D:Dec, B:Blk) = blk(D:Dec, B:Blk) .
  endm
```

6.4 Output

⟨*bplc-out*⟩≡
```
  mod OUT is
      ex DEC .

      sort OutAttrib .

      op out : -> OutAttrib [ctor] .
      op _:_ : OutAttrib ValueStack -> SemComp [ctor format(c! b! o o)] .
      op print : Exp -> Cmd [ctor format(! o)] .
      op PRINT : -> Control [ctor] .

      var ... : Set{SemComp} .

      op out : Conf -> NeValueStack [ctor] .
      eq out(< out : V:NeValueStack , ... >) = V:NeValueStack .

      eq [print] :
          < cnt : (print(E:Exp) C:ControlStack), ... > =
          < cnt : (E:Exp PRINT C:ControlStack), ... > [variant] .

      eq [print] :
          < cnt : (PRINT C:ControlStack),
            val : val(R:Rat) V:ValueStack,
            out : O:ValueStack, ... > =
          < cnt : C:ControlStack,
            val : V:ValueStack,
            out : val(R:Rat) O:ValueStack , ... > [variant] .
  endm
```

6.5 Abnormal termination

⟨*bplc-ext*⟩≡
```
  mod COMMAND-SEQ-AND-ABNORMAL-TERMINATION is
      ex OUT .

      sorts ExcAttrib Exc .

      -- Sequences had to be moved "down" from CMD to this module because
      -- the evaluation of the next command only takes place if no
      -- exit was executed.
      op seq : Cmd Cmd -> Cmd [format(! o)] .

      op exit : Exp -> Cmd [format(! o)] .
      op EXT : -> Exc .
      op EXIT : -> Control .
      op CNT : -> Exc .
      op exc : -> ExcAttrib .
      op _:_ : ExcAttrib Exc -> SemComp [format(c! b! o o)] .

      var ... : Set{SemComp} . var E : Env . var S : Store .
      var C : ControlStack . var V : ValueStack .

      eq [exit-cmd] :
          < cnt : exit(X:Exp) C, ... > = < cnt : X:Exp EXIT C, ... > [variant] .

      -- Maybe define a flush operation that sets the semantic components
      -- to their identity values.
      eq [exit-cmd] :
         < cnt : EXIT C,
           env : E,
           sto : S,
           val : A:Value V,
           out : O:ValueStack,
           locs : SL:Set{Loc},
           exc : CNT, ... > =
        < cnt : ecs,
           env : noEnv,
           sto : noStore,
           val : evs,
           out : A:Value,
           locs : noLocs,
           exc : EXT, ... > [variant] .

      -- Sequences had to be moved "down" to this module because
      -- the evaluation of the next command only takes place if no
```

```
                -- exit was executed.
        eq [seq-cmd] :
            < cnt : seq(C1:Cmd, C2:Cmd) C, exc : CNT, ... > =
            < cnt : C1:Cmd C2:Cmd C, exc : CNT, ... > [variant] .

        eq [seq-cmd] :
            < cnt : seq(C1:Cmd, C2:Cmd) C, exc : EXT, ... > =
            < cnt : ecs, exc : EXT, ... > [variant] .
    endm
```

6.6 Running a command

⟨*bplc*⟩≡
```
  mod BPLC is
      ex COMMAND-SEQ-AND-ABNORMAL-TERMINATION .

      var ... : Set{SemComp} .

      op getValue : Id Conf -> Storable .
      eq getValue(I:Id,
            < env : (I:Id |-> bind(L:Loc), E:Env) ,
            sto : (L:Loc |-> S:Storable, S:Store) ,
            ... >) = S:Storable .

      op run : Cmd Dec -> Conf .
      rl run(C:Cmd, D:Dec) =>
          < cnt : blk(D:Dec, C:Cmd) ecs,
            env : noEnv, sto : noStore , val : evs ,
            locs : noLocs , out : evs , exc : CNT > .
  endm
```

7 Model checking

⟨*bplc-mc*⟩≡
```
mod BPLC+META-LEVEL is
        pr BPLC . pr META-LEVEL .
endm

load model-checker

mod BPLC-MODEL-CHECKER is
    ex BPLC .
    pr MODEL-CHECKER .
        pr LTL-SIMPLIFIER .
        pr META-LEVEL .

    subsort Conf < State .

    var ... : Set{SemComp} .

    op valueOf : Id Rat Conf -> Bool .
    op valueOf : Id Bool Conf -> Bool .
    eq valueOf(I:Id, R:Rat,
          < env : (I:Id |-> bind(L:Loc), E:Env) ,
            sto : (L:Loc |-> store(R:Rat), S:Store) ,
            ... >) = true .
    eq valueOf(I:Id, R:Rat, C:Conf) = false [owise] .

    eq valueOf(I:Id, B:Bool,
          < env : (I:Id |-> bind(L:Loc), E:Env) ,
            sto : (L:Loc |-> store(B:Bool), S:Store) ,
            ... >) = true .
    eq valueOf(I:Id, B:Bool, C:Conf) = false [owise] .

    op _,_|=?_ : Dec Cmd Formula -> ModelCheckResult .
    eq D:Dec, C:Cmd |=? F:Formula =
          modelCheck(run(C:Cmd,D:Dec), F:Formula) .

    op makeOpDeclSet : Dec -> OpDeclSet .
    eq makeOpDeclSet(ref(idn(Q:Qid), E:Exp)) =
          (op Q:Qid : 'Rat -> 'Prop [none] .) .
    eq makeOpDeclSet(dec(D1:Dec, D2:Dec)) =
          makeOpDeclSet(D1:Dec) makeOpDeclSet(D2:Dec) .
    eq makeOpDeclSet(D:Dec) = none [owise] .

    op makeEqSet : Dec -> EquationSet .
    eq makeEqSet(ref(idn(Q:Qid), E:Exp)) =
```

40

```
                    (ceq '_|=_['S:State, Q:Qid['R:Rat]] = 'true.Bool
                     if 'valueOf['idn[upTerm(Q:Qid)],'R:Rat,'S:State] = 'true.Bool [none] .) .
        eq makeEqSet(dec(D1:Dec, D2:Dec)) =
                makeEqSet(D1:Dec) makeEqSet(D2:Dec) .
        eq makeEqSet(D:Dec) = none [owise] .

        op makeMCModule : Qid Dec -> SModule .
        eq makeMCModule(Q:Qid, D:Dec) =
            (mod qid("MODEL-CHECK-" + string(Q:Qid)) is
                            protecting 'BPLC-MODEL-CHECKER .
                sorts none .
                none -- SubsortDeclSet
                -- op 'bid : 'Qid -> 'GSLIdentifiers [none] .
                makeOpDeclSet(D:Dec)
                none -- MembAxSet
                makeEqSet(D:Dec)
                none endm) .
endm
```

8 Generating Python code

⟨*bplc-compile-to-python*⟩≡
```
mod COMPILE-TO-PYTHON is
    pr BPLC .
    pr QID-LIST .
    pr META-LEVEL .

    ops level incr : -> Nat .
    eq level = 3 .
    eq incr = 3 .

    op printSpaces : Nat -> QidList .
    eq printSpaces(0) = (nil).QidList .
    eq printSpaces(N:Nat) = '\s printSpaces(sd(N:Nat, 1)) .

    op id2Qid : Id -> QidList .
    eq id2Qid(idn(Q:Qid)) = Q:Qid .

    op formals2QidList : Formals -> QidList .
    eq formals2QidList(vod) = (nil).QidList .
    eq formals2QidList(par(I:Id)) = id2Qid(I:Id) .
    eq formals2QidList(for(F1:Formals, F2:Formals)) =
        formals2QidList(F1:Formals) ', formals2QidList(F2:Formals) .

    op blk2Block : Blk Dec Nat -> QidList .
    ceq blk2Block(blk(D1:Dec, C:Cmd), D2:Dec, N:Nat) =
        '\s ':
        declareGlobal(C:Cmd, D2:Dec, N:Nat + incr)
        dec2Decl(D1:Dec, D2:Dec, N:Nat + incr)
        cmd2Comm(C:Cmd, D2:Dec, N:Nat + incr)
     if declareGlobal(C:Cmd, D2:Dec, N:Nat + incr) =/= (nil).QidList .
    ceq blk2Block(blk(D1:Dec, C:Cmd), D2:Dec, N:Nat) =
         '\s ': '\n
         dec2Decl(D1:Dec, D2:Dec, N:Nat + incr)
         cmd2Comm(C:Cmd, D2:Dec, N:Nat + incr)
     if declareGlobal(C:Cmd, D2:Dec, N:Nat + incr) == (nil).QidList .
    ceq blk2Block(blk(C:Cmd), D:Dec, N:Nat) =
        '\s ':
        declareGlobal(C:Cmd, D:Dec, N:Nat + incr)
        cmd2Comm(C:Cmd, D:Dec, N:Nat + incr)
     if declareGlobal(C:Cmd, D:Dec, N:Nat + incr) =/= (nil).QidList .
    ceq blk2Block(blk(C:Cmd), D:Dec, N:Nat) =
         '\s ':
         cmd2Comm(C:Cmd, D:Dec, N:Nat + incr)
     if declareGlobal(C:Cmd, D:Dec, N:Nat + incr) =/= (nil).QidList .
```

42

```
op dec2Decl : Dec Dec Nat -> QidList .
eq dec2Decl(cns(I:Id, E:Exp), D:Dec, N:Nat) =
   id2Qid(I:Id) '= exp2Expr(E:Exp) .
eq dec2Decl(ref(I:Id, E:Exp), D:Dec, N:Nat) =
   id2Qid(I:Id) '= exp2Expr(E:Exp) .
eq dec2Decl(prc(I:Id, B:Blk), D:Dec, N:Nat) =
   'def id2Qid(I:Id) ''( '')
   blk2Block(B:Blk, D:Dec, N:Nat + incr) .
eq dec2Decl(prc(I:Id, F:Formals, B:Blk), D:Dec, N:Nat) =
   'def id2Qid(I:Id) ''( formals2QidList(F:Formals) '')
   blk2Block(B:Blk, D:Dec, N:Nat + incr) .
eq dec2Decl(dec(D1:Dec, D2:Dec), D3:Dec, N:Nat) =
   dec2Decl(D1:Dec, D3:Dec, N:Nat) '\n
   dec2Decl(D2:Dec, D3:Dec, N:Nat) .

op isGlobal : Id Dec -> Bool [memo] .
eq isGlobal(I1:Id, ref(I2:Id, E:Exp)) = I1:Id == I2:Id .
eq isGlobal(I1:Id, cns(I2:Id, E:Exp)) = false .
eq isGlobal(I1:Id, prc(I2:Id, F:Formals, B:Blk)) = false .
eq isGlobal(I1:Id, prc(I2:Id, B:Blk)) = false .
eq isGlobal(I:Id, dec(D1:Dec, D2:Dec)) =
   isGlobal(I:Id, D1:Dec) or isGlobal(I:Id, D2:Dec) .

op exp2Expr : Exp -> QidList .
eq exp2Expr(I:Id) = id2Qid(I:Id) .
eq exp2Expr(rat(R:Rat)) =
   metaPrettyPrint(upModule('RAT, false), upTerm(R:Rat)) .
eq exp2Expr(boo(B:Bool)) =
   metaPrettyPrint(upModule('BOOL, false), upTerm(B:Bool)) .
eq exp2Expr(add(E1:Exp, E2:Exp)) =
   exp2Expr(E1:Exp) '+ exp2Expr(E2:Exp) .
eq exp2Expr(mul(E1:Exp, E2:Exp)) =
   exp2Expr(E1:Exp) '* exp2Expr(E2:Exp) .
eq exp2Expr(sub(E1:Exp, E2:Exp)) =
   exp2Expr(E1:Exp) '- exp2Expr(E2:Exp) .
eq exp2Expr(div(E1:Exp, E2:Exp)) =
   exp2Expr(E1:Exp) '/ exp2Expr(E2:Exp) .
eq exp2Expr(neg(E:Exp)) =
   'not ''( exp2Expr(E:Exp) '') .
eq exp2Expr(eq(E1:Exp, E2:Exp)) =
   exp2Expr(E1:Exp) '== exp2Expr(E2:Exp) .
eq exp2Expr(and(E1:Exp, E2:Exp)) =
   exp2Expr(E1:Exp) 'and exp2Expr(E2:Exp) .
eq exp2Expr(or(E1:Exp, E2:Exp)) =
```

```
        exp2Expr(E1:Exp) 'or exp2Expr(E2:Exp) .
    eq exp2Expr(gt(E1:Exp, E2:Exp)) =
        exp2Expr(E1:Exp) '> exp2Expr(E2:Exp) .
    eq exp2Expr(ge(E1:Exp, E2:Exp)) =
        exp2Expr(E1:Exp) '>= exp2Expr(E2:Exp) .
    eq exp2Expr(lt(E1:Exp, E2:Exp)) =
        exp2Expr(E1:Exp) '< exp2Expr(E2:Exp) .
    eq exp2Expr(le(E1:Exp, E2:Exp)) =
        exp2Expr(E1:Exp) '<= exp2Expr(E2:Exp) .

    op cmd2Comm : Cmd Dec Nat -> QidList .
    eq cmd2Comm(print(E:Exp), D:Dec, N:Nat) =
        'print ''( exp2Expr(E:Exp) '') .
    eq cmd2Comm(assign(I:Id, E:Exp), D:Dec, N:Nat) =
        id2Qid(I:Id) '= exp2Expr(E:Exp) .
    eq cmd2Comm(if(E:Exp, B1:Blk, B2:Blk), D:Dec, N:Nat) =
        'if exp2Expr(E:Exp)
        blk2Block(B1:Blk, D:Dec, N:Nat + incr)
        'else
        blk2Block(B2:Blk, D:Dec, N:Nat + incr) .
    eq cmd2Comm(loop(E:Exp, B:Blk), D:Dec, N:Nat) =
        'while exp2Expr(E:Exp)
        blk2Block(B:Blk, D:Dec, N:Nat) .
    eq cmd2Comm(seq(C1:Cmd, C2:Cmd), D:Dec, N:Nat) =
        printSpaces(N:Nat)
        cmd2Comm(C1:Cmd, D:Dec, N:Nat) '\n
        printSpaces(N:Nat)
        cmd2Comm(C2:Cmd, D:Dec, N:Nat) .

    op declareGlobal : Cmd Dec Nat -> QidList .
   ceq declareGlobal(assign(I:Id, E:Exp), D:Dec, N:Nat) =
        '\n printSpaces(N:Nat) 'global id2Qid(I:Id)
    if isGlobal(I:Id, D:Dec) .
    eq declareGlobal(seq(C1:Cmd, C2:Cmd), D:Dec, N:Nat) =
        declareGlobal(C1:Cmd, D:Dec, N:Nat) '\n
        declareGlobal(C2:Cmd, D:Dec, N:Nat) .
    eq declareGlobal(C:Cmd, D:Dec, N:Nat) = (nil).Qid [owise] .
endm
```

References

1. A. Arnold. *Finite Transition Systems: Semantics of Communicating Systems.* Prentice Hall, Upper Saddle River, New Jersey 07458, 1994.
2. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, New York, NY, USA, 1998.
3. F. Chalub. An implementation of Modular Structural Operational Semantics in Maude. Master's thesis, Universidade Federal Fluminense, 2005.
4. F. Chalub and C. Braga. Maude MSOS Tool. *Electronic Notes in Theoretical Computer Science*, 176(4):133–146, 2006. http://dx.doi.org/10.1016/j.entcs.2007.06.012.
5. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, Cambridge, MA, USA, 1999.
6. M. Clavel, F. Durán, S. Eker, S. Escobar, N. Martí-Oliet, P. Lincoln, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.7.1).* SRI International and University of Illinois at Urbana-Champaign, http://maude.cs.uiuc.edu/maude2-manual/, July 2016.
7. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - a High-performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic.* Springer-Verlag, Berlin, Heidelberg, 2007.
8. J. A. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs.* MIT Press, Cambridge, MA, USA, 1996.
9. R. Goldblatt. *Logics of time and computation*, volume 7 of *CSLI Lecture Notes.* Center for the Study of Language and Information, Stanford, CA. ISBN: 0-937073-94-6, second edition edition, 1992.
10. N. Martí-Oliet and J. Meseguer. *Handbook of Philosophical Logic*, volume 9, chapter Rewriting logic as a logical and semantic framework, pages 1–87. Kluwer Academic Publishers, P.O. Box 17, 3300 AA Dordrecht, the Netherlands, 2002.
11. J. Meseguer. Conditional rewriting as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, April 1992.
12. P. D. Mosses. Component-based description of programming languages. In *Proceedings of the 2008 International Conference on Visions of Computer Science: BCS International Academic Conference*, VoCS'08, pages 275–286, Swindon, UK, 2008. BCS Learning & Development Ltd.
13. P. D. Mosses. Fundamental concepts and formal semantics of programming languages. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.457.4959&rep=rep1&type=pdf, 2009.
14. G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004. Special issue on SOS.
15. G. Roşu. From conditional to unconditional rewriting. In J. L. Fiadeiro, P. D. Mosses, and F. Orejas, editors, *Recent Trends in Algebraic Development Techniques*, pages 218–233, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
16. P. Viry. Elimination of conditions. *J. Symb. Comput.*, 28(3):381–400, 1999.