Notes on Formal Compiler Construction with the π Framework

Christiano Braga

Instituto de Computação
Universidade Federal Fluminense
http://www.ic.uff.br/~cbraga

Alpha version of August 10, 2018

Contents

Chapter 1

Introduction

Compiler construction is considered an intimidating discipline in Computer Science and related courses. This is perhaps captured quite graphically by the cover of the standard book on the subject, the so-called “Dragon book" (Compilers: Principles, Techniques, and Tools [2]), by Alfred V. Aho, Jeffrey D. Ullman and later on with Ravi Sethi and Monica S. Lam. There are “red”, “green” and “purple dragon” editions, but the Dragon, representing how burdensome people think of the subject, is always there.

The author has been developing and applying a formal approach, called π, for compiler construction, aiming at a simple technique, that relies on basic mathematics and standard Computer Science courses, that could eventually ease compiler construction and help teaching the subject.

π is comprised by π lib, a set of programming languages constructs inspired by Peter Mosses' Component-Based Semantics [26] and π Automata, an automata-based formalism to describe the operational semantics of programming languages, that generalizes Gordon Plotkin's Interpreting Automata approach [30]. To write a compiler using π, one needs to transform the (abstract) syntax tree of a given language into a description in π lib. Then, one can execute, validate or have machine code using different formal tools developed for π lib, such as an interpreter, a model checker, or a code generator, implemented following the formal semantics of π lib given in terms of π Automata.

This manuscript introduces π, an automata-based semantic framework for formal compiler construction and its implementation in the Maude language. A Python implementation as a Jupyter notebook is also underway, to explore different compilation and validation techniques. In this manuscript, we will focus on π Automata for the *dynamic semantics* of programming languages, and its Maude implementation. For the moment, parsing and transformation to π lib depend on the particular framework used to implement π, Maude in this manuscript. Optimization is also left to an external framework, such as LLVM. In the foreseeable future we intend to cover all phases of the compiler construction process, in a formal way, based on π Automata.

The remainder of these notes is organized as follows. Chapter 2 recalls some preliminary material to the discussion of π Automata, subject of Chapter 3. The π Automata semantics of π lib is discussed in Chapter 4, together with its Maude implementation. Chapter 5 describes a Python implementation for a subset of π lib. Chapter 6 documents a compiler for the IMP language using the π Framework. Chapter 7 reports on a Maude implementation for an executable environment for the Abstract Machine Notation based on π. Chapter 8 describes some related work and Chapter 9 concludes this manuscript with the usual final remarks and indication of future work.

Chapter 2

Preliminaries

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 transition system [3] (TS) is a pair $\mathscr{T} = (S, \rightarrow)$, where S denotes the set of the states of the system, $\rightarrow \subseteq S \times S$ is the transition relation. Transition systems are the standard models of structural operational semantics (SOS) descriptions.[1] 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. Rule 2.1 presents the general form of the transition 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 a 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.

$$\frac{\rho' \vdash \bigwedge_{i=1}^{n} t_i, \sigma_i \Rightarrow t'_i, \sigma'_i}{\rho \vdash f(t_1, t_2, \ldots, t_n), \sigma \Rightarrow f(t'_1, t'_2, \ldots, t'_n), \sigma'} \text{ if } Cnd \tag{2.1}$$

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 $(\rho, g, \sigma) \Rightarrow (\rho, g', \sigma')$ then the construction of the transition system $\mathscr{T}$ from $\mathscr{M}$ becomes straightforward with $(\rho, g, \sigma) \in S$.

Model checking [14] 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 [21] 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.

[1] As a matter of fact, SOS has *labeled* transition systems as models with the set of labels denoting actions of the system. Labels are essential while modeling action synchronization in concurrent systems. Therefore, since we are not discussing concurrency primitives in this manuscript, considering the more liberal transition systems as models of SOS descriptions will not cripple the proposal of this manuscript.

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 [15] is a high-performance implementation of Rewriting Logic [24], a formalism for the specification of concurrent systems that has been shown to be able to represent quite naturally many logical and semantic frameworks [23].

Maude[2] is an algebraic programming language. A program in Maude is organized by modules, and every module has an initial algebra [20] 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 Church-Rosser and terminating [4]. 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, which are terms in the Σ-algebra modulo the set of E equations and 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 3.3.1 exemplifies how this feature is intensively used in the Maude implementation of the π Automata framework.

An equational theory $\mathscr{E} = (\Sigma, E \cup A)$ is declared in Maude as a *functional* module with concrete syntax '`fmod` $\mathscr{E}$ `is` I Σ E `endfm`', where I is the list of modules to be included by $\mathscr{E}$. A sort s in Σ is declared with syntax '`sort` s'. A subsort inclusion between s and s' is declared with syntax '`subsort` s `<` s''. An operation $o : \vec{s_i} \to s'$ in Σ is declared with syntax '`op` o `:` $\vec{s_i}$ `->` s' `[`A^*`]` `.`', where $\vec{s_i}$ is the domain of o denoted by the product of s_i sorts, $1 \leq i \leq n$, $n \in \mathbb{N}$, and A^* is a list of A attributes including '`assoc`', '`comm`', '`id:` c', '`idem`', that denote what their names imply, and c is a constructor operator. Equations are declared with syntax '`eq` $L = R$ `.`' where L and R are terms in $\mathscr{T}_{\Sigma/A}(X)$, the Σ-algebra with variables in X, modulo axioms A. Equations can be conditional, declared, essentially, with syntax '`ceq` $L = R$ `if` $\bigwedge_i^n L_i = R_i$ `.`', where $n \in \mathbb{N}$, and $L_i, R_i \in \mathscr{T}_{\Sigma/A}(X)$. A rewrite theory $\mathscr{R} = (\Sigma, E \cup A, R)$ is declared in Maude as a *system* module with concrete syntax '`mod` $\mathscr{R}$ `is` I Σ E R `endm`' where Σ and E are declared with the same syntax as for functional modules and rules in R are declared with syntax '`rl [l] :` L `=>` R `.`' where l is the rule label. As equations, rules can also be conditional, declared with syntax '`crl` L `=>` R `if` $\bigwedge_i^n L_i = R_i \wedge \bigwedge_j^m L_j$ `=>` R_j `.`', where L_j,

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

[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.

$R_j \in \mathcal{T}_{\Sigma/A}(X)$ and $m \in \mathbb{N}$.

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. Section 3.3.1 illustrates how this "gauge" is used in the Maude implementation of the π framework.

2.2.1 Maude as a meta-tool

One of the distinctive characteristics of Maude is the support to meta-programming. Meta-level applications in Maude, applications that use meta-programming, apply the so called *descent functions* [16, Ch.14]. Such functions use a representation of modules as terms in a *universal* theory, implemented in Maude as a system module called META-LEVEL in its prelude. 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. An interesting example of metaReduce is the invocation of the model checker at the meta-level: (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.
- 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.
- The descent function metaSearch looks for terms that match 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.

2.2.2 Writing a compiler in Maude

A compiler for programs in a (source) language L into programs in a (target) language L' can be written in Maude as a meta-level application $\mathscr{C}$. The main components of $\mathscr{C}$ are: (i) a context-free grammar for L, (ii) the abstract syntax of L, (iii) a parser for programs in L that generates abstract syntax trees for L, (iv) a transformer from the abstract syntax of a program in L into the abstract syntax of a program in L', (v) a pretty-printer for the abstract syntax of programs in L', and (vi) a command-line user-interface for interacting with the compiler .

The context-free grammar of a language $G = (V, T, P, S)$, where V is the set of variables or non-terminals, T is the set of terminals, P is the set of productions of the form $V \rightarrow \alpha$, with $\alpha \in (V \cup T)^*$, and $S \notin (V \cup T)$ the initial symbol, is represented, in Maude, as a functional module $\mathscr{G} = (\Sigma, \emptyset \cup A)$, that is, $E = \emptyset$, where, essentially, non-terminals are captured as sorts in Σ, non-terminal relations are

captured by subsort inclusion, also in Σ, and terminals are represented as operations with appropriate signature in Σ, possibly with properties, such as associativity, declared in A.

The parser for L is a meta-function in a functional module $\mathscr{P} = (\Sigma, E)$ in Maude that includes, at least, the (i) functional module denoting the grammar of L, (ii) the functional module denoting the abstract syntax of L and (iii) the functional module META-LEVEL. The set E of equations in $\mathscr{P}$ are defined by structural induction on the syntax of L *encoded as meta-terms in Maude*, that is, they are such that the left-hand side of an equation is a meta-term denoting a statement in L and its right-hand side is a term in the initial algebra of the functional module denoting the abstract syntax of L.

A transformer from the AST of L to the AST of L' is a meta-function in a functional module $\mathscr{T} = (\Sigma, E)$ including, at least, the functional modules for the AST for L and L' and the META-LEVEL. Each equation in E is such that: (i) its left-hand side is given by a term with variables in the initial algebra of the functional module representing the AST of L, and, similarly, (ii) its right-hand side denotes a term with variables in the initial algebra of the functional module representing the AST of L'. When L' is Maude itself, that is, the compiler generates *a rewrite theory representing the rewriting logic semantics of* L, the many tools available in Maude, such as the rewrite module axioms engine, narrowing and Linear Temporal Logic model checker, are "lifted" to programs in L through its rewriting logic semantics.

A pretty-printer for the AST of L' is a meta-function in a functional module $\mathscr{P}\mathscr{P} = (\Sigma, E)$ such that the equations in E produce a list of quoted identifiers from a term in the initial algebra of the functional module denoting the AST of L'.

Chapter 3

π Automata

3.1 Interpreting automata

In [30], 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 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 form $\Gamma = \mathit{Value\ Stack} \times \mathit{Memory} \times \mathit{Control\ Stack}$, where $\mathit{Value\ Stack} = (L(G))^*$ with $L(G)$ the language generated by G,[1] the set Memory is a finite map $\mathit{Var} \rightarrow_{fin} \mathit{Storable}$ with $\mathit{Var} \in V$ and $\mathit{Storable} \subseteq T^*$, and the elements of the $\mathit{Control\ Stack} = (L(G) \cup KW)^*$, 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 the CFG of a programming language $\mathscr{L}$ with arithmetic expressions, Boolean expressions and commands.

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

The values in the *Value Stack* are elements of the set $\mathbb{T} \cup \mathbb{N} \cup \mathit{Var} \cup \mathit{BExp} \cup \mathit{Com}$, 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

[1] There are some situations where one may need to push not only computed values but *code* as well into the value stack. One such situation is when a *loop* is being evaluated and both the loop's test and body are pushed in order to "reconstruct" the loop for the next iteration.

the set $(Com \cup BExp \cup AExp \cup KW)^*$, 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 postfix 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 $c(p_1, p_2, \ldots, p_n)$, and the *Memory*, should $c(p_1, p_2, \ldots, p_n) \in L(Com)$.

For the transition relation of $\mathscr{I}$, let us consider the rules for arithmetic sum expressions.

$$\langle S, M, n\ C\rangle \quad \Rightarrow \quad \langle n\ S, M, C\rangle \tag{3.1}$$

$$\langle S, M, (e_1{+}e_2)\ C\rangle \quad \Rightarrow \quad \langle S, M, e_1\ e_2\ +\ C\rangle \tag{3.2}$$

$$\langle n\ m\ S, M, +\ C\rangle \quad \Rightarrow \quad \langle (n+m)\ S, M, C\rangle \tag{3.3}$$

where e_i are metavariables for arithmetic expressions, and $n, m \in \mathbb{N}$. Rule 3.2 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 3.1. 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 [33] and later by Roşu in [31], 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 [30], Interpreting Automata evaluation uses the control stack to push the evaluation context, so to speak, to the configuration. Unconditional rewriting has also very desirable computational consequences, in particular in Maude, regarding executability and performance. Model checking, for instance, does not consider transitions (rewrites) in the conditions of rules. Also, narrowing does not work, for the time being, on conditional rules. Regarding performance, the combination of a proper use of equations instead of rules together with unconditional rules provides an effective search mechanism. The use of equations shortens the state space and unconditional rules do not create "scratch pad" rewrites performing only forward rewriting.

As an example, let us recall Rule 2.1, the general form of the rule for the recursive 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 a 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 2.1 is as follows,

$$\{f(t_1, t_2, \ldots, t_n)\ C, \rho, \sigma\} \Rightarrow \{t_1\, t_2 \ldots t_n f\ C, \rho, \sigma\} \text{ if } Cnd, \tag{3.4}$$

where C is the control stack. Note that recursion 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 3.4. Copies of the environment (such as ρ' in Rule 2.1) 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 the Modular SOS approach to the formal semantics of programming languages.

Formally, a π-automaton is an Interpreting Automaton where, given an abstract finite pre-order *Sem*, for semantics components, its configurations are defined by $\Gamma = \uplus_{i=1}^{n} Sem$, 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 arithmetic sum in π Automata look very similar to the ones from Interpreting Automata.

$$\{S, M, n\ C, \ldots\} \Rightarrow \{n\ S, M, C, \ldots\} \tag{3.5}$$

$$\{S, M, (e_1 + e_2)\ C, \ldots\} \Rightarrow \{S, M, e_1\ e_2 + C, \ldots\} \tag{3.6}$$

$$\{n\ m\ S, M, +\ C, \ldots\} \Rightarrow \{(n+m)\ S, M, C, \ldots\} \tag{3.7}$$

The ellipsis "…" [2] are adopted as notation for "don't care" semantic components, 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 for arithmetic sum in Interpreting Automata would require such reformulation while in π Automata 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. Semantic component composition is *monotonic*, as the addition of new semantic components does not affect the transition relation, that is, $x \Rightarrow y$ implies $g(x) \Rightarrow g(y)$, where $x, y \in \Gamma$ and g 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 as a Term Rewriting System [4] 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 (TRS). Let us first recall two basic properties of a reduction relation from [4, 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 [4, Ch. 11],

[2] 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 [11].

otherwise non-terminating systems become terminating when an algebraic property, such as commutativity, is incorporated into the rewriting process. Given a TRS $(A, \longrightarrow)$, 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 TRS 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 TRS, as each rule rewrites a set of semantic components.

3.3.1 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 disjoint union set of semantic components. The trivial view SemComp maps terms of sort Elt to terms of sort SemComp. Module GSMC then imports 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.

```
1  fmod SEMANTIC-COMPONENTS is sorts SemComp . endfm
2  view SemComp from TRIV to SEMANTIC-COMPONENTS is sort Elt to SemComp . endv
3  fmod GSMC is ex VALUE-STACK . ex MEMORY . ex CONTROL-STACK . ex ENV .
4      ex SET{SemComp} * (op empty to noSemComp) .
5      sorts Attrib Conf EnvAttrib StoreAttrib ControlAttrib ValueAttrib .
6      subsort EnvAttrib StoreAttrib ControlAttrib ValueAttrib < Attrib .
7      op <_> : Set{SemComp} -> Conf [format(c! c! c! o)] .
8      op env : -> EnvAttrib . --- Semantic components
9      op sto : -> StoreAttrib .
10     op cnt : -> ControlAttrib .
11     op val : -> ValueAttrib .
12     op _:_ : EnvAttrib Env -> SemComp [ctor format(c! b! o o)] .
13     op _:_ : StoreAttrib Store -> SemComp [ctor format(r! b! o o)] .
14     op _:_ : ControlAttrib ControlStack -> SemComp [ctor format(c! b! o o)] .
15     op _:_ : ValueAttrib ValueStack -> SemComp [ctor format(c! b! o o)] .
16 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. 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.

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 their character is functional and therefore are implemented as equations in Maude. For sum, in equation `add-exp1`[3] 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 eq [add-exp2] : < cnt : (ADD C:ControlStack),
4                   val : (val(R1:Rat) val(R2:Rat) SK:ValueStack), ... > =
5                 < cnt : C:ControlStack,
6                   val : (val(R1:Rat + R2:Rat) SK:ValueStack), ... > [variant] .
```

3.4 Model checking π Automata

Model checking (e.g. [14]) is perhaps the most popular formal method for the validation of concurrent systems. The fact that it is an *automata-based automated validation technique* makes it a nice candidate to join a simple framework for teaching language construction that also aims at validation, such as the one proposed in this manuscript.

This section recalls the syntax and semantics for (a subset of) Linear Temporal Logic, one of the Modal Logics used in model checking, and discusses how to use this technique to validate π Automata, only the necessary to follow Section 6.

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

$$\phi ::= \top \mid \bot \mid p \mid \neg(\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid (\Diamond\phi) \mid (\Box\phi)$$

where connectives $\Diamond$, $\Box$ are called *temporal modalities*. They denote "Future state" and "Globally (all future states)". There is a precedence among them given by: first unary modalities, in the following order $\neg$, $\Diamond$ and $\Box$, then binary modalities, in the following order, $\wedge, \vee$ and $\rightarrow$.

The standard models for Modal Logics (e.g. [21]) 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 $L : W \rightarrow 2^{AP}$ 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 $\tau = s_1 \rightarrow s_2 \rightarrow \ldots$ is an example. A *suffix* 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 $\tau = s_1 \rightarrow \ldots$ a path in $\mathscr{K}$. Satisfaction of an LTL formula ϕ in a path τ, denoted $\tau \models \varphi$ is defined as follows,

$$\tau \models \top, \quad \tau \not\models \bot, \quad \tau \models p \ \textit{iff}\ p \in L(s_1), \quad \tau \models \neg\phi \ \textit{iff}\ \tau \not\models \phi,$$
$\tau \models \phi_1 \wedge \phi_2$ iff $\tau \models \phi_1$ and $\tau \models \phi_2$,
$\tau \models \phi_1 \vee \phi_2$ iff $\tau \models \phi_1$ or $\tau \models \phi_2$,
$\tau \models \phi_1 \rightarrow \phi_2$ iff $\tau \models \phi_2$ whenever $\tau \models \phi_1$,
$\tau \models \Box\phi$ iff for all $i \geq 1, \tau^i \models \phi$,
$\tau \models \Diamond\phi$ iff there is some $i \geq 1, \tau^i \models \phi$.

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

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:

$$\begin{aligned} &\forall \sigma \in \mathit{Memory}, v \in \mathit{Index}(\sigma), r \in \mathit{Storable}, \\ &\langle \sigma, \ldots \rangle \models p_v(r) =_{def} (\sigma(v) = r), \end{aligned} \tag{3.8}$$

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, used in Section 6, the LTL formula $\Box\neg[p_1(\mathit{crit}) \wedge p_2(\mathit{crit})]$ specifies safety ("nothing bad happens"), in this case both p_1 and p_2 in the critical section, 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 $\Box[p_1(\mathit{try}) \rightarrow \Diamond(p_1(\mathit{crit}))]$ specifies liveness ("something good eventually happens"), by stating that if a process, p_1 in this case, tries to enter the critical section it will eventually do so.

Chapter 4

π lib: Basic Programming Language Constructs

π lib is a subset of Constructive MSOS [28], as implemented in [9, Ch. 6]. In Section 4.1, π lib constructions are presented. Section 4.2 describes its Maude implementation. Their π-automata semantics is discussed in Section 4.3 and a simple compiler for the IMP language in Maude, using π lib, is described in Section 6.

4.1 π lib signature

The signature of π lib is organized in five parts: (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.

Grammars 4.1, 4.2, 4.3, and 4.4 declare the CFG for π lib expressions, commands and declarations. Given their simplicity, expressions and commands should be self-explanatory.

⟨*Control*⟩ ::= ⟨*Exp*⟩ | 'ADD' | 'SUB' | 'MUL' | 'DIV'

⟨*Exp*⟩ ::= ⟨*Id*⟩ | ⟨*AExp*⟩ | ⟨*BExp*⟩

⟨*Id*⟩ ::= 'idn' ⟨*ID*⟩

⟨*AExp*⟩ ::= ⟨*RAT*⟩ | ⟨*AOp*⟩ ⟨*AExp*⟩ ⟨*AExp*⟩

⟨*AOp*⟩ ::= 'add' | 'sub' | 'mul' | 'div'

Grammar 4.1: π lib arithmetic expressions

As for declarations, cns declares a constant and ref declares a variables. There are prc declarations for both parameterless procedures and parameterized ones. Lists of declarations are constructed with operator dec. Formal parameters are declared as vod, the meaningless parameter, or with par. Lists of formal parameters are constructed with keyword for. A block of commands may be declared as a pair of declarations and commands or simply as a commands. A procedure call is constructed by a cal parameterized by the procedure to be called. A cal operation may also be parameterized.

⟨*Control*⟩ ::= 'LT' | 'LE' | 'EQ' | 'NEG' | 'AND' | 'OR'

⟨*BExp*⟩ ::= 'boo' ⟨*Bool*⟩ | ⟨*BOp*⟩ ⟨*Exp*⟩ ⟨*Exp*⟩

⟨*BOp*⟩ ::= 'gt' | 'ge' | 'lt' | 'le' | 'eq' | 'neg' | 'and' | 'or'

Grammar 4.2: π lib Boolean expressions

⟨*Control*⟩ ::= ⟨*Cmd*⟩ | 'ASSIGN' | 'LOOP' | 'IF'

⟨*Cmd*⟩ ::= 'nop' | 'choice' ⟨*Cmd*⟩ ⟨*Cmd*⟩ |
'assign' ⟨*Id*⟩ ⟨*Exp*⟩ | 'loop' ⟨*BExp*⟩ ⟨*Cmd*⟩

Grammar 4.3: π lib commands

⟨*Control*⟩ ::= ⟨*Dec*⟩ | 'CNS' | 'REF' | 'CAL' | 'BLK' | 'FRE'

⟨*Dec*⟩ ::= 'cns' ⟨*Id*⟩ ⟨*Exp*⟩ | 'ref' ⟨*Id*⟩ ⟨*Exp*⟩ | 'prc' ⟨*Id*⟩ ⟨*Blk*⟩ | 'prc' ⟨*Id*⟩ ⟨*Formals*⟩ ⟨*Blk*⟩ | 'dec' ⟨*Dec*⟩ ⟨*Dec*⟩

⟨*Formal*⟩ ::= 'vod' | 'par' ⟨*Id*⟩

⟨*Formals*⟩ ::= ⟨*Formal*⟩ | 'for' ⟨*Formals*⟩ ⟨*Formals*⟩

⟨*Blk*⟩ ::= 'blk' ⟨*Cmd*⟩ | 'blk' ⟨*Dec*⟩ ⟨*Cmd*⟩

⟨*Cmd*⟩ ::= 'cal' ⟨*Id*⟩ | 'cal' ⟨*Id*⟩ ⟨*Actuals*⟩

⟨*Actuals*⟩ ::= 'act' ⟨*Actuals*⟩ ⟨*Actuals*⟩

Grammar 4.4: π lib declarations

4.2 π lib signature in Maude

We only discuss the π lib signature for arithmetic expressions. The remaining declarations follow a similar pattern. First, it module EXP includes modules QID, RAT, and GSMC, for quoted identifiers, rational numbers and Generalized SMC machines, respectively. Modules QID and RAT are part of the 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. Arithmetic and Boolean operations alike are declared as Maude operators, and so are elements of set *KW*.

```
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)] .

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

Listing 2: Signature for π lib expressions in Maude

Listing 3 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*.

```
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] .
    ...
endfm
```

Listing 3: 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 4. 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 8 in Section 4.3 gives more details on the semantics of block evaluation.)

```
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 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)] .
    ...
endm
```

Listing 4: 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.

```
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] .
    ...
endm
```

Listing 5: 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.

```
mod COMMAND-SEQ-AND-ABNORMAL-TERMINATION is
    ex OUT .

    sorts ExcAttrib Exc .

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

Listing 6: Signature for π lib abnormal termination in Maude

4.3 π Automata transitions for π lib dynamic semantics in Maude

From the EXP module, equations for Identifier and sum evaluation are described in Listing 7. 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 properties 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.

```
    ...
    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] .
    ...
```

Listing 7: π Automata equations for variable evaluation in Maude

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.

```
eq [loop] :
        < cnt : loop(E:Exp, K:Cmd) C:ControlStack, val : V:ValueStack, ... > =
        < cnt : E:Exp LOOP C:ControlStack, val : val(loop(E:Exp, K:Cmd)) V:ValueStack, ... >
[variant] .
rl [loop] :
       < cnt : LOOP C:ControlStack, val : val(true) val(loop(E:Exp, K:Cmd)) V:ValueStack, ... > =>
       < cnt : K:Cmd loop(E:Exp, K:Cmd) C:ControlStack, val : V:ValueStack, ... > [narrowing] .
```

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

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.

```
...
eq [blk-1] :
     < 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 ,
       locs : noLocs , ... > [variant] .

eq [blk-2] :
    < 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] .
...
```

Listing 8: π-automaton rules for block

Listing 9 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 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.)

```
...
eq [cal-1] :
     < cnt : cal(I:Id, A:Actuals) C, ... > =
     < cnt : I:Id A:Actuals CAL C, ... > [variant] .

eq [cal-2] :
     < 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 [prc-id] :
     < cnt : (I:Id C),
       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 9: π-automaton rules for function call

Chapter 5

π^2: π Framework in Python

5.1 π lib Expressions

5.1.1 Grammar for π lib Expressions

$$
\begin{array}{rcl}
Statement & ::= & Exp \\
Exp & ::= & ArithExp \mid BoolExp \\
ArithExp & ::= & \texttt{Sum}(Exp,Exp) \mid \texttt{Sub}(Exp,Exp) \mid \texttt{Mul}(Exp,Exp) \\
BoolExp & ::= & \texttt{Eq}(Exp,Exp) \mid \texttt{Not}(Exp)
\end{array}
$$

5.1.2 π lib Expressions in Python

We encode BNF rules as classes in Python. Every non-terminal gives rise to a class. The reduction relation ::= is encoded as inheritance. Operands are encoded as cells in a list object attribute, whose types are enforced by `assert` predicates on `isinstace` calls. The operand list `opr` is declared in the `Statement` class, whose constructor initializes the `opr` attribute with as many parameters as the subclass constructor might have.

```
In [1]: # π lib
        ## Statement
        class Statement:
            def __init__(self, *args):
                self.opr = args
            def __str__(self):
                ret = str(self.__class__.__name__)+"("
                for o in self.opr:
                    ret += str(o)
                ret += ")"
                return ret

        ## Expressions
        class Exp(Statement): pass
        class ArithExp(Exp): pass
        class Num(ArithExp):
            def __init__(self, f):
                assert(isinstance(f, (int, long)))
```

```
            ArithExp.__init__(self,f)
        class Sum(ArithExp):
            def __init__(self, e1, e2):
                assert(isinstance(e1, Exp) and isinstance(e2, Exp))
                ArithExp.__init__(self, e1, e2)
        class Sub(ArithExp):
            def __init__(self, e1, e2):
                assert(isinstance(e1, Exp) and isinstance(e2, Exp))
                ArithExp.__init__(self, e1, e2)
        class Mul(ArithExp):
            def __init__(self, e1, e2):
                assert(isinstance(e1, Exp) and isinstance(e2, Exp))
                ArithExp.__init__(self, e1, e2)
        class BoolExp(Exp): pass
        class Eq(BoolExp):
            def __init__(self, e1, e2):
                assert(isinstance(e1, Exp) and isinstance(e2, Exp))
                BoolExp.__init__(self, e1, e2)
        class Not(BoolExp):
            def __init__(self, e):
                assert(isinstance(e, Exp))
                BoolExp.__init__(self, e)
```

```
In [2]: exp = Sum(Num(1), Mul(Num(2), Num(4)))
        print(exp)
```

```
Sum(Num(1)Mul(Num(2)Num(4)))
```

However, if we create an ill-formed tree, an exception is raised.

```
exp2 = Mul(2.0, 1.0)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-3-de6e358a117c> in <module>()
----> 1 exp2 = Mul(2.0, 1.0)

<ipython-input-1-09d2d91ef407> in __init__(self, e1, e2)
     28 class Mul(ArithExp):
     29     def __init__(self, e1, e2):
---> 30         assert(isinstance(e1, Exp) and isinstance(e2, Exp))
     31         super().__init__(e1, e2)
     32 class BoolExp(Exp): pass

AssertionError:
```

5.1.3 π Automaton for π lib Expressions

The π automaton for π lib Expressions is implemented in the `ExpPiAut` class. Instances of `ExpPiAut` are dictionaries, that come initialized with two entries: one for the value stack, at index `val`, and

antother for the control stack, indexed `cnt`.

```
class ExpPiAut(dict):
    def __init__(self):
        self["val"] = ValueStack()
        self["cnt"] = ControlStack()
# ...
```

Class `ExpπAut` encapsulates the encoding for π lib Expression rules as private methods that are called by the public (polymorphic) `eval` method. In the following code snippet it calls the function that evaluates a `Sum` expression.

```
def eval(self):
    e = self.popCnt()
    if isinstance(e, Sum):
        self.__evalSum(e)
# ...
```

We use Maude syntax to specify π Automaton rules. This is the π rule for the evaluation of (floating point) numbers, described as an equation in Maude. It specifies that whenever a number is in the top of the control stack `C` is should be popped from `C` and pushed into the value stack `SK`.

```
   maude eq [num-exp] :     < cnt : (num(f:Float) C:ControlStack), val :
SK:ValueStack, ... >   =     < cnt : C:ControlStack,       val : (val(f:Float)
SK:ValueStack), ... > .
```

π rule `num-exp` is encoded in function `__evalNum(self, n)`. It receives a `Num` object in `n` whose sole attribute has the floating point number that `n` denotes. Method `pushVal(.)` pushes the given argument into the value stack.

```
def __evalNum(self, n):
    f = n.opr[0]
    self.pushVal(f)
```

5.1.4 The complete π Automaton for π lib Expressions in Python

```
In [3]: ## Expressions
        class ValueStack(list): pass
        class ControlStack(list): pass
        class ExpKW:
            SUM = "#SUM"
            SUB = "#SUB"
            MUL = "#MUL"
            EQ = "#EQ"
            NOT = "#NOT"
        class ExpPiAut(dict):
            def __init__(self):
                self["val"] = ValueStack()
                self["cnt"] = ControlStack()
            def val(self):
                return self["val"]
            def cnt(self):
```

```
        return self["cnt"]
    def pushVal(self, v):
        vs = self.val()
        vs.append(v)
    def popVal(self):
        vs = self.val()
        v = vs[len(vs) - 1]
        vs.pop()
        return v
    def pushCnt(self, e):
        cnt = self.cnt()
        cnt.append(e)
    def popCnt(self):
        cs = self.cnt()
        c = cs[len(cs) - 1]
        cs.pop()
        return c
    def emptyCnt(self):
        return len(self.cnt()) == 0
    def __evalSum(self, e):
        e1 = e.opr[0]
        e2 = e.opr[1]
        self.pushCnt(ExpKW.SUM)
        self.pushCnt(e1)
        self.pushCnt(e2)
    def __evalSumKW(self, e):
        v1 = self.popVal()
        v2 = self.popVal()
        self.pushVal(v1+v2)
    def __evalMul(self, e):
        e1 = e.opr[0]
        e2 = e.opr[1]
        self.pushCnt(ExpKW.MUL)
        self.pushCnt(e1)
        self.pushCnt(e2)
    def __evalMulKW(self):
        v1 = self.popVal()
        v2 = self.popVal()
        self.pushVal(v1*v2)
    def __evalSub(self, e):
        e1 = e.opr[0]
        e2 = e.opr[1]
        self.pushCnt(ExpKW.SUB)
        self.pushCnt(e1)
        self.pushCnt(e2)
    def __evalSubKW(self):
        v1 = self.popVal()
        v2 = self.popVal()
        self.pushVal(v1-v2)
```

```
        def __evalNum(self, n):
            f = n.opr[0]
            self.pushVal(f)
        def __evalEq(self, e):
            e1 = e.opr[0]
            e2 = e.opr[1]
            self.pushCnt(ExpKW.EQ)
            self.pushCnt(e1)
            self.pushCnt(e2)
        def __evalEqKW(self):
            v1 = self.popVal()
            v2 = self.popVal()
            self.pushVal(v1 == v2)
        def __evalNot(self, e):
            e = e.opr[0]
            self.pushCnt(ExpKW.NOT)
            self.pushCnt(e)
        def __evalNotKW(self):
            v = self.popVal()
            self.pushVal(not v)
        def eval(self):
            e = self.popCnt()
            if isinstance(e, Sum):
                self.__evalSum(e)
            elif e == ExpKW.SUM:
                self.__evalSumKW(e)
            elif isinstance(e, Sub):
                self.__evalSub(e)
            elif e == ExpKW.SUB:
                self.__evalSubKW()
            elif isinstance(e, Mul):
                self.__evalMul(e)
            elif e == ExpKW.MUL:
                self.__evalMulKW()
            elif isinstance(e, Num):
                self.__evalNum(e)
            elif isinstance(e, Eq):
                self.__evalEq(e)
            elif e == ExpKW.EQ:
                self.__evalEqKW()
            elif isinstance(e, Not):
                self.__evalNot(e)
            elif e == ExpKW.NOT:
                self.__evalNotKW()
            else:
                raise Exception("Ill formed: ", e)
```

```
In [4]: ea = ExpPiAut()
        print(exp)
        ea.pushCnt(exp)
```

```
        while not ea.emptyCnt():
            ea.eval()
            print(ea)
```

```
Sum(Num(1)Mul(Num(2)Num(4)))
{'cnt': ['#SUM', <__main__.Num instance at 0x10f4e57a0>,
                         <__main__.Mul instance at 0x10f4e5488>], 'val': []}
{'cnt': ['#SUM', <__main__.Num instance at 0x10f4e57a0>, '#MUL',
                          <__main__.Num instance at 0x10f4e5290>,
                          <__main__.Num instance at 0x10f4e55f0>], 'val': []}
{'cnt': ['#SUM', <__main__.Num instance at 0x10f4e57a0>, '#MUL',
                          <__main__.Num instance at 0x10f4e5290>], 'val': [4]}
{'cnt': ['#SUM', <__main__.Num instance at 0x10f4e57a0>, '#MUL'], 'val': [4, 2]}
{'cnt': ['#SUM', <__main__.Num instance at 0x10f4e57a0>], 'val': [8]}
{'cnt': ['#SUM'], 'val': [8, 1]}
{'cnt': [], 'val': [9]}
```

5.2 π lib Commands

5.2.1 Grammar for π lib Commands

$$
\begin{array}{rcl}
\mathit{Statement} & ::= & \mathit{Cmd} \\
\mathit{Exp} & ::= & \texttt{Id}(\mathit{String}) \\
\mathit{Cmd} & ::= & \texttt{Assign}(\mathit{Id},\mathit{Exp}) \mid \texttt{Loop}(\mathit{BoolExp},\mathit{Cmd}) \mid \texttt{CSeq}(\mathit{Cmd},\mathit{Cmd})
\end{array}
$$

Commands are language constructions that require both an environement and a memory store to be evaluated. From a syntactic standpoint, they extend statements and expressions, as an identifier is an expression.

5.2.2 Grammar for π lib Commands in Python

The enconding of the grammar for commands follows the same mapping of BNF rules as classes we used for expressions.

```
In [5]: ## Commands
        class Cmd(Statement): pass
        class Id(Exp):
            def __init__(self, s):
                assert(isinstance(s, str))
                Exp.__init__(self, s)
        class Assign(Cmd):
            def __init__(self, i, e):
                assert(isinstance(i, Id) and isinstance(e, Exp))
                Cmd.__init__(self, i, e)
        class Loop(Cmd):
            def __init__(self, be, c):
                assert(isinstance(be, BoolExp) and isinstance(c, Cmd))
                Cmd.__init__(self, be, c)
        class CSeq(Cmd):
            def __init__(self, c1, c2):
```

```
                assert(isinstance(c1, Cmd) and isinstance(c2, Cmd))
                Cmd.__init__(self, c1, c2)

In [6]: cmd = Assign(Id("x"), Num(1))
        print(type(cmd))
        print(cmd)

<type 'instance'>
Assign(Id(x)Num(1))
```

5.2.3 Complete π Automaton for Commands in Python

```
In [7]: ## Commands
        class Env(dict): pass
        class Loc(long): pass
        class Sto(dict): pass
        class CmdKW:
            ASSIGN = "#ASSIGN"
            LOOP = "#LOOP"
        class CmdPiAut(ExpPiAut):
            def __init__(self):
                self["env"] = Env()
                self["sto"] = Sto()
                ExpPiAut.__init__(self)
            def env(self):
                return self["env"]
            def getLoc(self, i):
                en = self.env()
                return en[i]
            def sto(self):
                return self["sto"]
            def updateStore(self, l, v):
                st = self.sto()
                st[l] = v
            def __evalAssign(self, c):
                i = c.opr[0]
                e = c.opr[1]
                self.pushVal(i.opr[0])
                self.pushCnt(CmdKW.ASSIGN)
                self.pushCnt(e)
            def __evalAssignKW(self):
                v = self.popVal()
                i = self.popVal()
                l = self.getLoc(i)
                self.updateStore(l, v)
            def __evalId(self, i):
                s = self.sto()
                l = self.getLoc(i)
                self.pushVal(s[l])
```

```
    def __evalLoop(self, c):
        be = c.opr[0]
        bl = c.opr[1]
        self.pushVal(Loop(be, bl))
        self.pushVal(bl)
        self.pushCnt(CmdKW.LOOP)
        self.pushCnt(be)
    def __evalLoopKW(self):
        t = self.popVal()
        if t:
            c = self.popVal()
            lo = self.popVal()
            self.pushCnt(lo)
            self.pushCnt(c)
        else:
            self.popVal()
            self.popVal()
    def __evalCSeq(self, c):
        c1 = c.opr[0]
        c2 = c.opr[1]
        self.pushCnt(c2)
        self.pushCnt(c1)
    def eval(self):
        c = self.popCnt()
        if isinstance(c, Assign):
            self.__evalAssign(c)
        elif c == CmdKW.ASSIGN:
            self.__evalAssignKW()
        elif isinstance(c, Id):
            self.__evalId(c.opr[0])
        elif isinstance(c, Loop):
            self.__evalLoop(c)
        elif c == CmdKW.LOOP:
            self.__evalLoopKW()
        elif isinstance(c, CSeq):
            self.__evalCSeq(c)
        else:
            self.pushCnt(c)
            ExpPiAut.eval(self)
```

5.2.4 π lib Declarations

5.2.5 Grammar for π lib Declarations

$$
\begin{array}{rcl}
\mathit{Statement} & ::= & \mathit{Dec} \\
\mathit{Exp} & ::= & \texttt{Ref}(\mathit{Exp}) \mid \texttt{Cns}(\mathit{Exp}) \\
\mathit{Cmd} & ::= & \texttt{Blk}(\mathit{Dec}, \mathit{Cmd}) \\
\mathit{Dec} & ::= & \texttt{Bind}(\mathit{Id}, \mathit{Exp}) \mid \texttt{DSeq}(\mathit{Dec}, \mathit{Dec})
\end{array}
$$

5.2.6 Grammar for π lib Declarations in Python

```
In [8]: ## Declarations
        class Dec(Statement): pass
        class Bind(Dec):
            def __init__(self, i, e):
                assert(isinstance(i, Id) and isinstance(e, Exp))
                Dec.__init__(self, i, e)
        class Ref(Exp):
            def __init__(self, e):
                assert(isinstance(e, Exp))
                Exp.__init__(self, e)
        class Cns(Exp):
            def __init__(self, e):
                assert(isinstance(e, Exp))
                Exp.__init__(self, e)
        class Blk(Cmd):
            def __init__(self, d, c):
                assert(isinstance(d, Dec) and isinstance(c, Cmd))
                Cmd.__init__(self, d, c)
        class DSeq(Dec):
            def __init__(self, d1, d2):
                assert(isinstance(d1, Dec) and isinstance(d2, Dec))
                Dec.__init__(self, d1, d2)
```

5.2.7 Complete π Automaton for π lib Declarations in Python

```
In [9]: ## Declarations
        class DecExpKW(ExpKW):
            REF = "#REF"
            CNS = "#CNS"
        class DecCmdKW(CmdKW):
            BLKDEC = "#BLKDEC"
            BLKCMD = "#BLKCMD"
        class DecKW:
            BIND = "#BIND"
            DSEQ = "#DSEQ"
        class DecPiAut(CmdPiAut):
            def __init__(self):
                self["locs"] = []
                CmdPiAut.__init__(self)
            def locs(self):
                return self["locs"]
            def pushLoc(self, l):
                ls = self.locs()
                ls.append(l)
            def __evalRef(self, e):
                ex = e.opr[0]
                self.pushCnt(DecExpKW.REF)
                self.pushCnt(ex)
```

```
def __newLoc(self):
    sto = self.sto()
    if sto:
        return max(list(sto.keys())) + 1
    else:
        return 0.0
def __evalRefKW(self):
    v = self.popVal()
    l = self.__newLoc()
    self.updateStore(l, v)
    self.pushLoc(l)
    self.pushVal(l)
def __evalBind(self, d):
    i = d.opr[0]
    e = d.opr[1]
    self.pushVal(i)
    self.pushCnt(DecKW.BIND)
    self.pushCnt(e)
def __evalBindKW(self):
    l = self.popVal()
    i = self.popVal()
    x = i.opr[0]
    self.pushVal({x:l})
def __evalDSeq(self, ds):
    d1 = ds.opr[0]
    d2 = ds.opr[1]
    self.pushCnt(DecKW.DSEQ)
    self.pushCnt(d2)
    self.pushCnt(d1)
def __evalDSeqKW(self):
    d2 = self.popVal()
    d1 = self.popVal()
    d1.update(d2)
    self.pushVal(d1)
def __evalBlk(self, d):
    ld = d.opr[0]
    c = d.opr[1]
    l = self.locs()
    self.pushVal(list(l))
    self.pushVal(c)
    self.pushCnt(DecCmdKW.BLKDEC)
    self.pushCnt(ld)
def __evalBlkDecKW(self):
    d = self.popVal()
    c = self.popVal()
    l = self.locs()
    self.pushVal(l)
    en = self.env()
    ne = en.copy()
```

```
            ne.update(d)
            self.pushVal(en)
            self["env"] = ne
            self.pushCnt(DecCmdKW.BLKCMD)
            self.pushCnt(c)
        def __evalBlkCmdKW(self):
            en = self.popVal()
            ls = self.popVal()
            self["env"] = en
            s = self.sto()
            s = {k:v for k,v in s.items() if k not in ls}
            self["sto"] = s
            #del ls
            ols = self.popVal()
            self["locs"] = ols
        def eval(self):
            d = self.popCnt()
            if isinstance(d, Bind):
                self.__evalBind(d)
            elif d == DecKW.BIND:
                self.__evalBindKW()
            elif isinstance(d, DSeq):
                self.__evalDSeq(d)
            elif d == DecKW.DSEQ:
                self.__evalDSeqKW()
            elif isinstance(d, Ref):
                self.__evalRef(d)
            elif d == DecExpKW.REF:
                self.__evalRefKW()
            elif isinstance(d, Blk):
                self.__evalBlk(d)
            elif d == DecCmdKW.BLKDEC:
                self.__evalBlkDecKW()
            elif d == DecCmdKW.BLKCMD:
                self.__evalBlkCmdKW()
            else:
                self.pushCnt(d)
                CmdPiAut.eval(self)
```

Factorial example

```
In [10]: dc = DecPiAut()
         fac = Loop(Not(Eq(Id("y"), Num(0))),
                 CSeq(Assign(Id("x"), Mul(Id("x"), Id("y"))),
                      Assign(Id("y"), Sub(Id("y"), Num(1)))))
         dec = DSeq(Bind(Id("x"), Ref(Num(1))),
                    Bind(Id("y"), Ref(Num(200))))
         fac_blk = Blk(dec, fac)
         dc.pushCnt(fac_blk)
```

```
        while not dc.emptyCnt():
            aux = dc.copy()
            dc.eval()
            if dc.emptyCnt():
                print(aux)
```

```
{'locs': [0.0, 1.0], 'cnt': [], 'sto': {0.0: 78865786736479050355236321393218506229513597768717326329474253324435944996340334292030428401198462390417721213891963883025764279024263710506192662495282993111346285727076331723739698894392244562145166424025403329186413122742829485327752424240757390324032125740557956866022603190417032406235170085879617892222278962370389737472000000000000000000000000000000000000000000000000L, 1.0: 0}, 'env': {'y': 1.0, 'x': 0.0}, 'val': []}
```

Chapter 6

A π compiler for IMP in Maude

In this Section, the use of π lib is illustrated by a compiler for a simple (and yet Turing-complete) imperative language called IMP. 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.

A π 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 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 language; (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. Each component is discussed next, but pretty-printing.

6.1 IMP syntax

⟨*program*⟩ ::= '`module`' ⟨*ident*⟩ ⟨*clauses*⟩ '`end`'

⟨*clauses*⟩ ::= ⟨*var*⟩$^?$ ⟨*const*⟩$^?$ ⟨*init*⟩$^?$ ⟨*proc*⟩$^+$

⟨*var*⟩ := '`var`' ⟨*ident*⟩$^+$

⟨*const*⟩ := '`const`' ⟨*ident*⟩$^+$

⟨*init*⟩ := '`init`' ⟨*ini*⟩$^+$

⟨*ini*⟩ := ⟨*ident*⟩ '`=`' ⟨*exp*⟩

⟨*proc*⟩ := '`proc`' ⟨*ident*⟩ '`(`' ⟨*ident*⟩* '`)`' ⟨*block*⟩

⟨*block*⟩ := '`{`' ⟨*cmd*⟩$^+$ '`}`'

⟨*cmd*⟩ := ⟨*ident*⟩ ':=' ⟨*exp*⟩
| ⟨*if*⟩ | ⟨*while*⟩ | ⟨*print*⟩ | ⟨*exit*⟩
| ⟨*call*⟩ |⟨*seq*⟩ | ⟨*choice*⟩

⟨*if*⟩ := 'if' ⟨*boolexp*⟩ ⟨*cmd*⟩ 'else' ⟨*cmd*⟩
| 'if' ⟨*boolexp*⟩ ⟨*cmd*⟩ 'else' ⟨*block*⟩
| 'if' ⟨*boolexp*⟩ ⟨*block*⟩ 'else' ⟨*cmd*⟩
| 'if' ⟨*boolexp*⟩ ⟨*block*⟩ 'else' ⟨*block*⟩

⟨*while*⟩ := 'while' ⟨*boolexp*⟩ ⟨*block*⟩

⟨*print*⟩ := 'print' '(' ⟨*exp*⟩ ')'

⟨*exit*⟩ := 'exit' '(' ⟨*exp*⟩ ')'

⟨*call*⟩ := ⟨*ident*⟩ '(' ⟨*exp*⟩* ')'

⟨*seq*⟩ := ⟨*cmd*⟩ ';' ⟨*cmd*⟩

⟨*choice*⟩ := ⟨*cmd*⟩ '|' ⟨*cmd*⟩

⟨*exp*⟩ := ⟨*ident*⟩
| ⟨*ident*⟩ ⟨*arithop*⟩ ⟨*ident*⟩
| ⟨*boolexp*⟩

⟨*arithop*⟩ := '+' | '-' | '*' | '|' | '/'

⟨*boolexp*⟩ := ⟨*ident*⟩ ⟨*boolop*⟩ ⟨*ident*⟩

⟨*boolop*⟩ := '~' | '==' | '<' | '<=' | '>' | '>='

6.2 IMP's command-line interface

In Listing 10 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. 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. 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

```
< getTerm(T:ResultPair?) ; compileMod(getTerm(T:ResultPair?)) ;
  '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.

```
mod IMP-INTERFACE is
 ...
 op <_;_;_> : MetaIMPModule Dec? QidList -> IMPState .
 vars QIL QIL' QIL'' QIL1 QIL2 : QidList .
 --- Loading a module.
 crl [in] : ['module Q:Qid QIL, < M:MetaIMPModule ; D:Dec? ; QIL' >, QIL''] =>
            if (T:ResultPair? :: ResultPair) then
                [nil, < getTerm(T:ResultPair?) > ;
                        compileMod(getTerm(T:ResultPair?)) > ;
                        'IMP: '\b 'Module Q:Qid 'loaded. '\o >, QIL'']
            else [nil, < noModule ; noDec ; nil >,
                    printParseError('module Q:Qid QIL, T:ResultPair?)]
            fi
 if T:ResultPair? :=
    metaParse(upModule('IMP-GRAMMAR, false), 'module Q:Qid QIL, 'ModuleDecl) .
 ...
endm
```

Listing 10: IMP's command-line interface in Maude

6.3 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 10, that implements IMP's read-eval-loop. Essentially, variables (or non-terminals) are represented by sorts, terminals by constants and grammar rules are represented by operations. Module IMP-GRAMMAR 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 ExpressionDecl, 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. Attributes `prec` and `gather` are declared for disambiguation.

```
fmod IMP-GRAMMAR is pr TOKEN . pr RAT .
 inc PREDICATE-DECL . inc COMMAND-DECL .
 sorts VariablesDecl ConstantsDecl OperationsDecl ProcDeclList
       ProcDecl FormalsDecl BlockCommandDecl ExpressionDecl
       InitDecl InitDeclList InitDecls ClausesDecl
       ModuleDecl Expression .
 subsort InitDecl < InitDeclList .
 subsort VariablesDecl ConstantsDecl ProcDeclList InitDecls < ClausesDecl .
 subsort BlockCommandDecl < CommandDecl .
 subsort ProcDecl < ProcDeclList .
 subsort PredicateDecl < ExpressionDecl .
 op _+_ : Token Token -> ExpressionDecl [gather(e E) prec 15] . --- Arithmetic expressions
 op _+_ : Token ExpressionDecl -> ExpressionDecl [gather(e E) prec 15] .
 op _+_ : ExpressionDecl Token -> ExpressionDecl [gather(e E) prec 15] .
 op _+_ : ExpressionDecl ExpressionDecl -> ExpressionDecl [gather(e E) prec 15] .
...
endfm
```

6.4 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. [1] 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. 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 an IMP 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 meta-level descent function downTerm to help us parse them into proper constants.

```
op compileId : Qid -> Id .
eq compileId(I:Qid) = idn(downTerm(I:Qid, 'Qid)) .
op compileId : Term -> Id .
eq compileId('token[I:Qid]) =
   if (metaParse(upModule('RAT, false),
         downTerm(I:Qid, 'Qid), 'Rat) :: ResultPair)
   then rat(downTerm(getTerm(metaParse(upModule('RAT, false),
        downTerm(I:Qid, 'Qid), 'Rat)), 1/2))
   else
     if (metaParse(upModule('BOOL, false),
           downTerm(I:Qid, 'Qid), 'Bool) :: ResultPair)
     then boo(downTerm(getTerm(metaParse(upModule('BOOL, false),
       downTerm(I:Qid, 'Qid), 'Bool)), true))
     else idn(downTerm(I:Qid, 'Qid))
     fi
   fi .
```

To conclude this Section, the compilation of IMP arithmetic expressions simply maps them to their prefixed syntax counterpart in π lib, e.g, an IMP expression a + b is compiled to add(compileExp(a), compileExp(b)).

```
op compileExp : Term -> Exp .
ceq compileExp(I:Qid) = compileId(I:Qid) if not(I:Qid :: Constant) .
eq compileExp('token[I:Qid]) = compileId('token[I:Qid]) .
eq compileExp('_+_[T1:Term, T2:Term]) = add(compileExp(T1:Term), compileExp(T2:Term)) .
eq compileExp('_-_[T1:Term, T2:Term]) = sub(compileExp(T1:Term), compileExp(T2:Term)) .
eq compileExp('_*_[T1:Term, T2:Term]) = mul(compileExp(T1:Term), compileExp(T2:Term)) .
eq compileExp('_/_[T1:Term, T2:Term]) = div(compileExp(T1:Term), compileExp(T2:Term)) .
```

6.5 Complete IMP compiler in Maude

6.5.1 Parser

Module `TOKEN` specifies what `Tokens`, `TokenLists` and `NeTokenList` are. They are constructed with operators `token`, `bubble` and `neTokenList`, where attribute `Exclude` specify which terms are not to be constructed by the given constructor.

⟨*token-module*⟩≡

```
fmod TOKEN is
pr QID-LIST .
sorts Token Bubble TokenList NeTokenList .
subsort Token < TokenList .
```

[1] This is not the case for variable and constant declarations that require initializations to be mapped to a π lib ref declaration, a simple exercise but useful stimulate non bijectional mappings between the source language and π lib. A similar situation arises when compiling IMP code to Python, as variable declarations are local by default, requiring the `global` modifier otherwise. This also not be the case for other programming languages, such as the denotation of object-oriented constructions [9].

```
  op _,_ : TokenList TokenList -> TokenList [assoc prec 5] .

  op token : Qid -> Token
     [special
       (id-hook Bubble        (1 1)
        op-hook qidSymbol     (<Qids> : ~> Qid)
        id-hook Exclude       ( true false nop print ))] .

  op bubble : QidList -> Bubble
     [special
       (id-hook Bubble        (1 -1)
        op-hook qidListSymbol (__ : QidList QidList ~> QidList)
        op-hook qidSymbol     (<Qids> : ~> Qid)
        id-hook Exclude       ( | if then else end { }
                                while ; := = nop )) ] .

  op neTokenList : QidList -> NeTokenList
     [special
       (id-hook Bubble        (1 -1)
        op-hook qidListSymbol (__ : QidList QidList ~> QidList)
        op-hook qidSymbol     (<Qids> : ~> Qid)
        id-hook Exclude       ( . ))] .
endfm
```

Module IMP-GRAMMAR implements the grammar of Section 6.1 in Maude. Each sort implements a non-terminal of the grammar in Section 6.1 whereas constants, tokens, as described in module TOKEN, or operators implement terminals. For instance, the following rule in the grammar of Section 6.1

⟨*proc*⟩ := 'proc' ⟨*ident*⟩ '(' ⟨*ident*⟩* ')' ⟨*block*⟩

is implemented in Module IMP-GRAMMAR by the operation declarations

```
op proc__ : Token BlockCommandDecl -> ProcDecl [prec 50] .
op proc_`(_`)_ : Token TokenList BlockCommandDecl -> ProcDecl [prec 50] .
```

together with the declarations for sorts ProcDecl and BlockCommandDecl.

⟨*imp-grammar-module*⟩≡

```
fmod IMP-GRAMMAR is
 pr TOKEN .
 pr RAT .
 inc PREDICATE-DECL .
 inc COMMAND-DECL .
 sorts VariablesDecl ConstantsDecl OperationsDecl ProcDeclList
       ProcDecl FormalsDecl BlockCommandDecl ExpressionDecl
       InitDecl InitDeclList InitDecls ClausesDecl
       ModuleDecl Expression .

 subsort InitDecl < InitDeclList .
 subsort VariablesDecl ConstantsDecl ProcDeclList
         InitDecls < ClausesDecl .
 subsort BlockCommandDecl < CommandDecl .
 subsort ProcDecl < ProcDeclList .
 subsort PredicateDecl < ExpressionDecl .

 *** Boolean expressions
 op _==_ : Token Token -> PredicateDecl [prec 30] .
 op _==_ : Token ExpressionDecl -> PredicateDecl [prec 30] .
 op _==_ : ExpressionDecl Token -> PredicateDecl [prec 30] .
 op _==_ : ExpressionDecl ExpressionDecl -> PredicateDecl
           [prec 30] .
 op _/\_ : PredicateDecl PredicateDecl -> PredicateDecl
           [assoc comm prec 35] .
 op ~ : PredicateDecl -> PredicateDecl .

 *** Arithmetic expressions
 op _+_ : Token Token -> ExpressionDecl [gather(e E) prec 15] .
 op _+_ : Token ExpressionDecl -> ExpressionDecl [gather(e E) prec 15] .
 op _+_ : ExpressionDecl Token -> ExpressionDecl [gather(e E) prec 15] .
 op _+_ : ExpressionDecl ExpressionDecl -> ExpressionDecl
          [gather(e E) prec 15] .

 op _-_ : Token Token -> ExpressionDecl  [gather(e E) prec 15] .
 op _-_ : Token ExpressionDecl -> ExpressionDecl [gather(e E) prec 15] .
 op _-_ : ExpressionDecl Token -> ExpressionDecl [gather(e E) prec 15] .
 op _-_ : ExpressionDecl ExpressionDecl -> ExpressionDecl
```

```
          [gather(e E) prec 15] .

  op _*_ : Token Token -> ExpressionDecl [gather(e E) prec 10] .
  op _*_ : Token ExpressionDecl -> ExpressionDecl [gather(e E) prec 10] .
  op _*_ : ExpressionDecl Token -> ExpressionDecl [gather(e E) prec 10] .
  op _*_ : ExpressionDecl ExpressionDecl -> ExpressionDecl
           [gather(e E) prec 10] .

  op _/_ : Token Token -> ExpressionDecl [prec 20] .
  op _/_ : Token ExpressionDecl -> ExpressionDecl [prec 20] .
  op _/_ : ExpressionDecl Token -> ExpressionDecl [prec 20] .
  op _/_ : ExpressionDecl ExpressionDecl -> ExpressionDecl [prec 20] .

  *** Commands
  op _:=_ : Token Token -> CommandDecl [prec 40] .
  op _:=_ : Token ExpressionDecl -> CommandDecl [prec 40] .

  op _() : Token -> CommandDecl .
  op _(_) : Token Bubble -> CommandDecl .
  op print : Token -> CommandDecl .
  op print : ExpressionDecl -> CommandDecl .
  op _;_ : CommandDecl CommandDecl -> CommandDecl [assoc prec 50] .
  op _|_ : CommandDecl CommandDecl -> CommandDecl [assoc prec 50] .
  op while_do_ : PredicateDecl BlockCommandDecl -> CommandDecl [prec 40] .
  op {_} : CommandDecl -> BlockCommandDecl [prec 35] .
  op if__else_ : PredicateDecl CommandDecl CommandDecl -> CommandDecl
     [prec 40] .

  *** Declarations
  op module__end : Token ClausesDecl -> ModuleDecl [prec 80] .
  op __ : ClausesDecl ClausesDecl -> ClausesDecl [assoc comm prec 70] .
  op var_ : TokenList -> VariablesDecl [prec 60] .
  op const_ : TokenList -> ConstantsDecl [prec 60] .
  op init_ : InitDecl -> InitDecls [prec 60] .
  op init_ : InitDeclList -> InitDecls  [prec 60] .
  op _=_ : Token Token -> InitDecl  [prec 40] .
  op _=_ : Token ExpressionDecl -> InitDecl  [prec 40] .
  op _,_ : InitDeclList InitDeclList -> InitDeclList  [assoc prec 50] .
  op proc__ : Token BlockCommandDecl -> ProcDecl [prec 50] .
  op proc_`(_`)_ : Token TokenList BlockCommandDecl -> ProcDecl
     [prec 50] .
  op __ : ProcDeclList ProcDeclList -> ProcDeclList
    [assoc comm prec 60] .
endfm
```

6.5.2 Compiling to π lib

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

⟨*compile-imp-to-bplc*⟩≡
```
mod COMPILE-IMP-TO-BPLC is
 inc PREDICATE-DECL .
 inc COMMAND-DECL .
 pr BPLC .
 pr META-LEVEL .

 ⟨Compiling IMP expressions to BPLC Exp⟩
 ⟨Compiling IMP commands to BPLC Cmd⟩
 ⟨Compiling IMP declarations to BPLC Dec⟩
```

The compilation from IMP to π exp relates IMP tokens to π Id, IMP arithmetic and boolean expressions to π 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

⟨*Compiling tokens*⟩≡
```
op compileId : Qid -> Id .
eq compileId(I:Qid) = idn(downTerm(I:Qid, 'Qid)) .

op compileId : Term -> Id .
eq compileId('token[I:Qid]) =
   if (metaParse(upModule('RAT, false),
        downTerm(I:Qid, 'Qid), 'Rat)  :: ResultPair)
       then rat(downTerm(getTerm(metaParse(upModule('RAT, false),
            downTerm(I:Qid, 'Qid), 'Rat)), 1/2))
   else
    if (metaParse(upModule('BOOL, false),
         downTerm(I:Qid, 'Qid), 'Bool) :: ResultPair)
    then boo(downTerm(getTerm(metaParse(upModule('BOOL, false),
                 downTerm(I:Qid, 'Qid), 'Bool)), true))
    else idn(downTerm(I:Qid, 'Qid))
    fi
   fi .
```

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

⟨*Compiling arithmetic expressions*⟩≡

```
op compileExp : Term -> Exp .
ceq compileExp(I:Qid) = compileId(I:Qid)
 if not(I:Qid :: Constant) .
eq compileExp('token[I:Qid]) = compileId('token[I:Qid]) .
eq compileExp('_+_[T1:Term, T2:Term]) =
    add(compileExp(T1:Term), compileExp(T2:Term)) .
eq compileExp('_-_[T1:Term, T2:Term]) =
    sub(compileExp(T1:Term), compileExp(T2:Term)) .
eq compileExp('_*_[T1:Term, T2:Term]) =
    mul(compileExp(T1:Term), compileExp(T2:Term)) .
eq compileExp('_/_[T1:Term, T2:Term]) =
    div(compileExp(T1:Term), compileExp(T2:Term)) .
```

A treatment similar to arithmetic expressions is given to Boolean expressions, with a particular care with constants, as we did in «Compiling tokens». Here, however, we do not require a call to `downTerm` as the constants for `PredicateDecl` are already typed.

⟨*Compiling boolean expressions*⟩≡

```
eq compileExp('true.PredicateDecl) = boo(true) .
eq compileExp('false.PredicateDecl) = boo(false) .
eq compileExp('~[T:Term]) = neg(compileExp(T:Term)) .
eq compileExp('_/\_[T1:Term,T2:Term]) =
   and(compileExp(T1:Term), compileExp(T2:Term)) .
eq compileExp('_\/_[T1:Term,T2:Term]) =
   or(compileExp(T1:Term), compileExp(T2:Term)) .
eq compileExp('_==_[T1:Term,T2:Term]) =
   eq(compileExp(T1:Term), compileExp(T2:Term)) .
eq compileExp('_>_[T1:Term,T2:Term]) =
   gt(compileExp(T1:Term), compileExp(T2:Term)) .
eq compileExp('_>=_[T1:Term,T2:Term]) =
   ge(compileExp(T1:Term), compileExp(T2:Term)) .
eq compileExp('_<_[T1:Term,T2:Term]) =
   lt(compileExp(T1:Term), compileExp(T2:Term)) .
eq compileExp('_<=_[T1:Term,T2:Term]) =
   le(compileExp(T1:Term), compileExp(T2:Term)) .
```

⟨*Compiling IMP commands to BPLC Cmd*⟩≡

```
op compileCmd : Term -> Cmd .
eq compileCmd('nop.CommandDecl) = nop .
eq compileCmd('_:=_['token[I:Qid], T:Term]) =
   assign(compileId('token[I:Qid]), compileExp(T:Term)) .
eq compileCmd(''{_'}[T:Term]) = blk(compileCmd(T:Term)) .
eq compileCmd('_;_[T1:Term, T2:Term]) =
   seq(compileCmd(T1:Term), compileCmd(T2:Term)) .
eq compileCmd('_|_[T1:Term, T2:Term]) =
   choice(compileCmd(T1:Term), compileCmd(T2:Term)) .
eq compileCmd('if__else_[T1:Term, T2:Term, T3:Term]) =
   if(compileExp(T1:Term), compileCmd(T2:Term),
      compileCmd(T3:Term)) .
eq compileCmd('while_do_[T1:Term, T2:Term]) =
   loop(compileExp(T1:Term), compileCmd(T2:Term)) .
eq compileCmd('print[T:Term]) = print(compileId(T:Term)) .

*** Compiling IMP commands to BPLC Cmd:
*** Procedure calls deserve more attention...

eq compileCmd('_'(')['token[I:Qid]]) =
   cal(compileId('token[I:Qid])) .
eq compileCmd('_'(_')['token[I:Qid], 'bubble[Q:Qid]]) =
   cal(compileId('token[I:Qid]), compileActuals(Q:Qid)) .
eq compileCmd('_'(_')['token[I:Qid],
              'bubble['__[TL:TermList]]]) =
   cal(compileId('token[I:Qid]), compileActuals(TL:TermList)) .

op compileActuals : TermList -> Actuals .
eq compileActuals(Q:Qid) = compileId('token[Q:Qid]) .
eq compileActuals((TL1:TermList, ''',.Qid , TL2:TermList)) =
   act(makeExp(TL1:TermList) , compileActuals(TL2:TermList)) .
eq compileActuals(TL:TermList) = makeExp(TL:TermList) [owise] .

op makeExp : TermList -> Exp .
eq makeExp(Q:Qid) = compileId(Q:Qid) .
ceq makeExp(TL:TermList) =
    compileExp(
     getTerm(
      metaParse(upModule('IMP-GRAMMAR, false),
       makeExprDeclQidList(TL:TermList), 'ExpressionDecl)))
 if (metaParse(upModule('IMP-GRAMMAR, false),
            makeExprDeclQidList(TL:TermList), 'ExpressionDecl) ::
          ResultPair) .

op makeExprDeclQidList : TermList -> QidList .
eq makeExprDeclQidList((empty).TermList) = (nil).QidList .
ceq makeExprDeclQidList(Q:Qid) = downTerm(Q:Qid, 'error)
 if downTerm(Q:Qid, 'error) =/= 'error .
```

```
eq makeExprDeclQidList((Q:Qid , TL:TermList)) =
   makeExprDeclQidList(Q:Qid) makeExprDeclQidList(TL:TermList) .
```

⟨*Compiling IMP declarations to BPLC Dec*⟩≡

```
*** Compiling variables. The initialization clause is
*** necessary to properly declare variables according to BPLC
*** ref construct.

op compileVar : Term Term -> Dec .
op $compileVar : Term Term -> Dec .
eq compileVar('var_[TL1:TermList], 'init_[TL2:TermList]) =
   $compileVar(TL1:TermList, TL2:TermList) .
eq $compileVar('token[I:Qid], '_=_['token[I:Qid], T:Term]) =
   ref(compileId('token[I:Qid]), compileExp(T:Term)) .
eq $compileVar('token[I:Qid],
   ('_‘,_['_=_['token[I:Qid], T:Term], TL:TermList])) =
   ref(compileId('token[I:Qid]), compileExp(T:Term)) .
ceq $compileVar('token[I1:Qid],
    ('_‘,_['_=_['token[I2:Qid], T:Term], TL:TermList])) =
    $compileVar('token[I1:Qid], TL:TermList)
 if I1:Qid =/= I2:Qid .
eq $compileVar('_‘,_['token[I:Qid], IS:TermList],
               '_=_['token[I:Qid], T:Term]) =
   ref(compileId('token[I:Qid]), compileExp(T:Term)) .
eq $compileVar('_‘,_['token[I:Qid], IS:TermList],
              '_‘,_[TL:TermList]) =
   dec($compileVar('token[I:Qid], '_‘,_[TL:TermList]),
    $compileVar(IS:TermList, '_‘,_[TL:TermList])) .

*** Compiling constants.
op compileConst : Term Term -> Dec .
op $compileConst : Term Term -> Dec .
eq compileConst('const_[T1:Term], 'init_[T2:Term]) =
   $compileConst(T1:Term, T2:Term) .
eq $compileConst('token[I:Qid],
                 '_=_['token[I:Qid], T:Term]) =
   cns(compileId('token[I:Qid]), compileExp(T:Term)) .
eq $compileConst('token[I:Qid],
                ('_‘,_['_=_['token[I:Qid], T:Term], TL:TermList])) =
   cns(compileId('token[I:Qid]), compileExp(T:Term)) .
ceq $compileConst('token[I1:Qid],
                ('_‘,_['_=_['token[I2:Qid], T:Term], TL:TermList])) =
   $compileConst('token[I1:Qid], TL:TermList)
 if I1:Qid =/= I2:Qid .
eq $compileConst('_‘,_['token[I:Qid], IS:TermList],
                 '_=_['token[I:Qid], T:Term]) =
   cns(compileId('token[I:Qid]), compileExp(T:Term)) .
eq $compileConst('_‘,_['token[I:Qid], IS:TermList], '_‘,_[TL:TermList]) =
   dec($compileConst('token[I:Qid], '_‘,_[TL:TermList]),
    $compileConst(IS:TermList, '_‘,_[TL:TermList])) .

*** Compiling procedure declarations.
```

```
  op compileProc : TermList -> Dec .
  op compileToFormals : TermList -> Formals .
  eq compileProc('__[TL1:TermList, TL2:TermList]) =
     dec(compileProc(TL1:TermList), compileProc(TL2:TermList)) .
  eq compileProc('proc__['token[O:Qid], TL:TermList, T:Term]) =
     prc(compileId('token[O:Qid]), compileCmd(T:Term)) .
  eq compileProc('proc_‘(_‘)_['token[O:Qid], TL:TermList, T:Term]) =
     prc(compileId('token[O:Qid]),
      compileToFormals(TL:TermList), compileCmd(T:Term)) .
  eq compileToFormals('token[O:Qid]) = par(compileId('token[O:Qid])) .
  eq compileToFormals('_‘,_[TL1:TermList, TL2:TermList]) =
     for(compileToFormals(TL1:TermList), compileToFormals(TL2:TermList)) .

  *** Compiling modules.
  op compileMod : Term -> Dec .
  eq compileMod('module__end['token[I:Qid],
                '__['var_[T1:Term],
                '__['const_[T2:Term],
                '__['init_[T3:Term],
                                T:Term]]]]) =
     dec(compileVar('var_[T1:Term],'init_[T3:Term]),
      dec(compileConst('const_[T2:Term], 'init_[T3:Term]),
       compileProc(T:Term))) .
  eq compileMod('module__end['token[I:Qid],
                '__['var_[T1:Term],
                '__['init_[T3:Term],
                T:Term]]]) =
     dec(compileVar('var_[T1:Term],'init_[T3:Term]),
          compileProc(T:Term)) .
  eq compileMod('module__end['token[I:Qid],
                '__['var_[T1:Term],
                '__['const_[T2:Term],
                '__['init_[T3:Term],
                '__[TL:TermList]]]]]) =
     dec(compileVar('var_[T1:Term],'init_[T3:Term]),
      dec(compileConst('const_[T2:Term], 'init_[T3:Term]),
        compileProc('__[TL:TermList]))) .
  eq compileMod('module__end['token[I:Qid],
                '__['var_[T1:Term],
                '__['init_[T3:Term],
                '__[TL:TermList]]]]) =
     dec(compileVar('var_[T1:Term],'init_[T3:Term]),
          compileProc('__[TL:TermList])) .
endm
```

6.5.3 Pretty-printing IMP programs, their execution and model checking results

⟨*imp-pretty-printing*⟩≡

```
mod IMP-PRETTY-PRINTING is
 pr BPLC-MODEL-CHECKER .
 pr COMPILE-TO-PYTHON .
 pr INT .

 op printTermList : TermList -> QidList .
 eq printTermList(empty) = (nil).QidList .
 eq printTermList((Q:Qid,TL:TermList)) =
    downTerm(Q:Qid,'Qid) printTermList(TL:TermList) .

 op printTokens : Term -> QidList .
 eq printTokens(Q:Qid) = Q:Qid .
 eq printTokens('token[Q:Qid]) = downTerm(Q:Qid, 'Qid) .
 eq printTokens('_`,_[T1:Term, T2:Term]) =
    printTokens(T1:Term) printToken('`,) '\s printTokens(T2:Term) .

 *** Pretty-printing IMP Expressions
 op printExp : Term -> QidList .
 eq printExp('true.PredicateDecl) = 'true .
 eq printExp('false.PredicateDecl) = 'false .
 eq printExp('bubble['__[TL:TermList]]) = printTermList(TL:TermList) .
 eq printExp('token[Q:Qid]) = downTerm(Q:Qid, 'Qid) .
 eq printExp('_==_[T1:Term, T2:Term]) =
    printExp(T1:Term) printToken('==) printExp(T2:Term) .
 eq printExp('_\/_[T1:Term, T2:Term]) =
    printExp(T1:Term) printToken('\/) printExp(T2:Term) .
 eq printExp('_/\_[T1:Term, T2:Term]) =
    printExp(T1:Term) printToken('\/) printExp(T2:Term) .
 eq printExp('~[T:Term]) =
    printToken('~) printExp(T:Term) .
 eq printExp('_<_[T1:Term, T2:Term]) =
    printExp(T1:Term) printToken('<) printExp(T2:Term) .
 eq printExp('_<=_[T1:Term, T2:Term]) =
    printExp(T1:Term) printToken('<=) printExp(T2:Term) .
 eq printExp('_>_[T1:Term, T2:Term]) =
    printExp(T1:Term) printToken('>) printExp(T2:Term) .
 eq printExp('_>=_[T1:Term, T2:Term]) =
    printExp(T1:Term) printToken('>=) printExp(T2:Term) .
 eq printExp('_+_[T1:Term, T2:Term]) =
    printExp(T1:Term) printToken('+) printExp(T2:Term) .
 eq printExp('_*_[T1:Term, T2:Term]) =
    printExp(T1:Term) printToken('*) printExp(T2:Term) .
 eq printExp('_-_[T1:Term, T2:Term]) =
    printExp(T1:Term) printToken('-) printExp(T2:Term) .
 eq printExp('_/_[T1:Term, T2:Term]) =
    printExp(T1:Term) printToken('/) printExp(T2:Term) .
```

```
 eq printExp(T:Term) =
    metaPrettyPrint(upModule('IMP-GRAMMAR, false), T:Term) [owise] .

 *** Pretty-printing IMP Commands
 op printCmd : Term Nat -> QidList .
 eq printCmd('nop.CommandDecl, N:Nat) = 'nop .
 eq printCmd('print[T:Term], N:Nat) =
    printToken('print) printToken('`() printExp(T:Term) printToken('`)) .
 eq printCmd('_`(_`)[T1:Term, T2:Term], N:Nat) =
    printExp(T1:Term) printToken('`() printExp(T2:Term) printToken('`)) .
 eq printCmd('_|_[T1:Term, '_;_[T2:TermList]], N:Nat) =
    printSpaces(N:Nat)
    printCmd(T1:Term, N:Nat) printToken('|) '\n
    printSpaces(N:Nat) printToken('`()
    printCmd('_;_[T2:TermList], N:Nat) printToken('`)) .
ceq printCmd('_|_[T:TermList, F:Qid[TL:TermList]], N:Nat) =
    printSpaces(N:Nat) printCmd(T:TermList, N:Nat) printToken('|)
    printCmd(F:Qid[TL:TermList], N:Nat)
 if F:Qid =/= '_;_ .
ceq printCmd('_;_[F1:Qid[TL1:TermList], F2:Qid[TL2:TermList]], N:Nat) =
    printCmd(F1:Qid[TL1:TermList], N:Nat) '\s printToken(';)
    printCmd(F2:Qid[TL2:TermList], N:Nat)
 if F1:Qid =/= '_;_ and F2:Qid =/= '_;_ .
ceq printCmd('_;_[F1:Qid[TL1:TermList], F2:Qid[TL2:TermList]], N:Nat) =
    printSpaces(N:Nat)
    printCmd(F1:Qid[TL1:TermList], N:Nat) '\s printToken(';) '\n
    printSpaces(N:Nat)
    printCmd(F1:Qid[TL1:TermList], N:Nat)
 if F1:Qid == '_;_ or F2:Qid == '_;_ .
 eq printCmd('_:=_[T1:Term, T2:Term], N:Nat) =
    printExp(T1:Term) printToken(':=) printExp(T2:Term) .
 eq printCmd('while_do_[T1:Term, T2:Term], N:Nat) =
    printToken('while) ' printToken('`() printExp(T1:Term) printToken('`)) '
    printToken('do)
    printCmd(T2:Term, N:Nat) .
 eq printCmd('if__else_[T1:Term, T2:Term, T3:Term], N:Nat) =
    printToken('if) ' printToken('`() printExp(T1:Term) printToken('`)) '
    printCmd(T2:Term, N:Nat)
    '\s printToken('else) printCmd(T3:Term, N:Nat) .
 eq printCmd('`{_`}[C:Constant], N:Nat) =
    '\s printToken('`{)
    printCmd(C:Constant, 0)
    printToken('`}) .
 eq printCmd('`{_`}['_;_[TL:TermList]], N:Nat) =
    '\s printToken('`{) '\n
    printSpaces(N:Nat + incr)
    printCmd('_;_[TL:TermList], N:Nat + incr) '\n
    printSpaces(N:Nat) printToken('`}) .
 eq printCmd('`{_`}['_|_[TL:TermList]], N:Nat) =
```

```
      '\s printToken(''{) '\n
      printCmd('_|_[TL:TermList], N:Nat + incr) '\n
      printSpaces(N:Nat) printToken(''}) .
  ceq printCmd(''{_'}[F:Qid[TL:TermList]], N:Nat) =
      '\s printToken(''{) '\n
      printSpaces(N:Nat + incr)
      printCmd(F:Qid[TL:TermList], N:Nat + incr) '\n
      printSpaces(N:Nat) printToken(''})
   if F:Qid =/= '_;_ /\ F:Qid =/= '_|_ .

   eq printCmd(T:Term, N:Nat) =
      printSpaces(N:Nat)
      metaPrettyPrint(upModule('IMP-GRAMMAR, false), T:Term) [owise] .

   *** Pretty-printing IMP Declarations
   op printModule : Term -> QidList .
   eq printModule('module__end[T1:Term, T2:Term]) =
      printToken('module) printTokens(T1:Term)
          printClauses(T2:Term)
          '\n printToken('end) .

   op printClauses : Term -> QidList .
   eq printClauses('proc__[T1:Term, T2:Term]) =
      '\n printSpaces(level)
      printToken('proc) printTokens(T1:Term)
      printCmd(T2:Term, level) .
   eq printClauses('__['proc__[T1:Term, T2:Term], T3:Term]) =
      '\n printSpaces(level) printToken('proc) printTokens(T1:Term)
      printCmd(T2:Term, level)
      printClauses(T3:Term) .
   eq printClauses('proc_'(_')_[T1:Term, T2:Term, T3:Term]) =
      '\n printSpaces(level)
      printToken('proc) printTokens(T1:Term)
      printToken(''() printTokens(T2:Term) printToken(''))
      printCmd(T3:Term, level) .
   eq printClauses('__['proc_'(_')_[T1:Term, T2:Term, T3:Term], T4:Term]) =
      '\n printSpaces(level) printToken('proc)
      printTokens(T1:Term) printToken(''() printTokens(T2:Term) printToken(''))
      printCmd(T3:Term, level)
      printClauses(T4:Term) .
   eq printClauses('__['init_[T1:Term], T2:Term]) =
      '\n printSpaces(level) printToken('init)
      printInit(T1:Term) printClauses(T2:Term) .
   eq printClauses('__['const_[T1:Term], T2:Term]) =
      '\n printSpaces(level) printToken('const)
      printTokens(T1:Term) printClauses(T2:Term) .
   eq printClauses('__['var_[T1:Term], T2:Term]) =
          '\n printSpaces(level) printToken('var)
      printTokens(T1:Term) printClauses(T2:Term) .
```

```
  op printInit : Term -> QidList .
  eq printInit('_=_[T1:Term, T2:Term]) =
     printTokens(T1:Term) printToken('=) printExp(T2:Term) .
  eq printInit('_‘,_[T1:Term, T2:Term]) =
     printInit(T1:Term) printToken('‘,) '\s printInit(T2:Term) .

  *** Pretty-printing parse error
  op printQidList : QidList -> QidList .
  eq printQidList(nil) = (nil).QidList .
  eq printQidList(Q:Qid QL:QidList) = printToken(Q:Qid) printQidList(QL:QidList) .

  op printToken : Qid -> Qid .
  eq printToken(Q:Qid) = '\b '\! Q:Qid '\o .

  op printInputWithError : QidList Nat -> QidList .
  op $printInputWithError : QidList Int QidList -> QidList .
  eq printInputWithError(QL:QidList, N:Nat) =
     $printInputWithError(QL:QidList, N:Nat, (nil).QidList) .
  eq $printInputWithError(nil, I:Int, QL:QidList) = QL:QidList .
  eq $printInputWithError(QL:QidList, 0, (nil).QidList) = (nil).QidList .
 ceq $printInputWithError(QL1:QidList, 0, (QL2:QidList Q:Qid)) =
     QL2:QidList '\r '\u '»> Q:Qid '«< 'HERE '\o
     printQidList(QL1:QidList)
  if QL2:QidList =/= nil .
  eq $printInputWithError(Q:Qid QL1:QidList, I:Int, QL2:QidList) =
     $printInputWithError(QL1:QidList, (I:Int + (- 1)), QL2:QidList
         printToken(Q:Qid)) .

  op printParseError : QidList ResultPair? -> Qid .
  eq printParseError('module Q:Qid QL:QidList , noParse(N:Nat)) =
     'IMP: 'Error 'at 'position
     metaPrettyPrint(upModule('NAT, false), upTerm(N:Nat))
         'while 'parsing 'Module Q:Qid ': '\n
         printInputWithError(('module Q:Qid QL:QidList), N:Nat) .
  eq printParseError(QL:QidList, ambiguity(T1:ResultPair, T2:ResultPair)) =
     'IMP: '\r 'Ambiguous 'parse '\o '\n
     metaPrettyPrint(upModule('IMP-GRAMMAR, false), getTerm(T1:ResultPair)) '\n
     '\r 'vs. '\o '\n
     metaPrettyPrint(upModule('IMP-GRAMMAR, false), getTerm(T2:ResultPair)) .

  *** Pretty-print exec and mc output
  op printState : Env Store -> QidList .
  eq printState(noEnv, S:Store) = (nil).QidList .
  eq printState((I:Id |-> bind(L:Loc)) , ((L:Loc |-> store(R:Rat)), S:Store)) =
     printId(I:Id) '= metaPrettyPrint(upModule('RAT, false), upTerm(R:Rat)) .
 ceq printState(((I:Id |-> bind(L:Loc)), E:Env) ,
     ((L:Loc |-> store(R:Rat)), S:Store)) =
     printId(I:Id) '=
```

```
    metaPrettyPrint(upModule('RAT, false), upTerm(R:Rat))
    printState(E:Env , (L:Loc |-> store(R:Rat), S:Store))
 if E:Env =/= noEnv .
 eq printState((I:Id |-> bind(L:Loc)), (L:Loc |-> store(B:Bool), S:Store)) =
    printId(I:Id) '= metaPrettyPrint(upModule('BOOL, false), upTerm(B:Bool)) .
ceq printState(((I:Id |-> bind(L:Loc)), E:Env) ,
    ((L:Loc |-> store(B:Bool)), S:Store)) =
        printId(I:Id) '= metaPrettyPrint(upModule('RAT, false), upTerm(B:Bool)) '‘,
        printState(E:Env , (L:Loc |-> store(B:Bool), S:Store))
 if E:Env =/= noEnv .
 eq printState(((I:Id |-> B:Bindable), E:Env) , S:Store) =
    printState(E:Env , S:Store) [owise] .

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

 op printCnt : ControlStack ValueStack -> QidList .
 eq printCnt((LOOP C:ControlStack),
            (V:Value val(loop(E:Exp, K:Cmd)) VS:ValueStack)) =
    printToken('while) printToken('‘() printExp(E:Exp) printToken('‘)) '... .
 eq printCnt((choice(K1:Cmd, K2:Cmd) C:ControlStack), VS:ValueStack) =
    printCmd(K1:Cmd) printToken('|) printCmd(K2:Cmd) .
 eq printCnt(C:ControlStack, VS:ValueStack) =
    '\n 'Constrol 'stack:
    metaPrettyPrint(upModule('BPLC+META-LEVEL, false),
     upTerm(C:ControlStack))
    '\n 'Value 'stack:
     metaPrettyPrint(upModule('BPLC+META-LEVEL, false),
     upTerm(VS:ValueStack)) .

 op printExp : Exp -> QidList .
 eq printExp(rat(R:Rat)) = '\g metaPrettyPrint(upModule('RAT, false),
     upTerm(R:Rat)) '\o .
 eq printExp(boo(B:Bool)) = '\g metaPrettyPrint(upModule('BOOL, false),
     upTerm(B:Bool)) '\o .
 eq printExp(neg(E:Exp)) = printToken('~) printExp(E:Exp) .
 eq printExp(and(E1:Exp, E2:Exp)) =
    printExp(E1:Exp) printToken('/\) printExp(E2:Exp) .
 eq printExp(eq(E1:Exp, E2:Exp)) =
    printExp(E1:Exp) printToken('==) printExp(E2:Exp) .
 eq printExp(idn(Q:Qid)) = printTokens(Q:Qid) .
 eq printExp(E:Exp) = metaPrettyPrint(upModule('BPLC, false), upTerm(E:Exp)) .

 op printCmd : Cmd -> QidList .
 eq printCmd(seq(C1:Cmd, C2:Cmd)) =
    printCmd(C1:Cmd) printToken(';) printCmd(C2:Cmd) .
 eq printCmd(assign(I:Id, E:Exp)) =
    printExp(I:Id) printToken(':=) printExp(E:Exp) .
 eq printCmd(if(E:Exp, C1:Cmd, C2:Cmd)) =
```

```
    printToken('if) printExp(E:Exp) '... .
  eq printCmd(choice(C1:Cmd, C2:Cmd)) = printCmd(C1:Cmd)
    printToken('|) printCmd(C2:Cmd) .
  eq printCmd(C:Cmd) = metaPrettyPrint(upModule('BPLC, false), upTerm(C:Cmd)) .

  op printConf : Conf -> QidList .
  eq printConf(< env : E:Env , sto : S:Store , exc : X:Exc ,
                cnt : C:ControlStack, val : V:ValueStack, ...:Set{SemComp} > ) =
    if X:Exc == CNT
    then printCnt(C:ControlStack, V:ValueStack) '`[ printState(E:Env, S:Store) '`]
        else 'IMP: 'Internal 'error 'while 'printing 'configuration.
    fi .

  op printValueStack : ValueStack -> QidList .
  eq printValueStack(evs) = '\b 'No 'output. '\o .
  eq printValueStack(val(R:Rat) evs) =
    metaPrettyPrint(upModule('RAT, false), upTerm(R:Rat)) .
  eq printValueStack(val(B:Bool) evs) =
    metaPrettyPrint(upModule('BOOL, false), upTerm(B:Bool)) .
  eq printValueStack(V:Value VS:ValueStack) =
    printValueStack(V:Value) printValueStack(VS:ValueStack) .

  op printOut : Conf -> QidList .
  eq printOut(< out : O:ValueStack , ...:Set{SemComp} > ) =
    printValueStack(O:ValueStack) .

  op printExec : Term ~> QidList .
 ceq printExec(T:Term) = printOut(downTerm(T:Term, < noSemComp >))
  if downTerm(T:Term, < noSemComp >) =/= < noSemComp > .

  op printTraceStep : TraceStep -> QidList .
  eq printTraceStep({T:Term, Y:Type, R:Rule}) =
    printConf(downTerm(T:Term, < noSemComp >)) .

  op printTrace : Trace Nat -> QidList .
  eq printTrace(nil, N:Nat) = (nil).QidList .
  eq printTrace(TS:TraceStep, N:Nat) =
    '\b 'State
    metaPrettyPrint(upModule('NAT, false),
     upTerm(N:Nat)) ': '\o '\n
    printTraceStep(TS:TraceStep) .
 ceq printTrace(TS:TraceStep T:Trace, N:Nat) =
    '\b 'State
    metaPrettyPrint(upModule('NAT, false),
     upTerm(N:Nat)) ': '\o '\n
    printTraceStep(TS:TraceStep) '\n printTrace(T:Trace, (N:Nat + 1))
  if T:Trace =/= nil .

  op printTransitionList : TransitionList -> QidList .
```

```
 eq printTransitionList(nil) = (nil).QidList .
 eq printTransitionList({C:Conf, R:RuleName}) = printConf(C:Conf) .
ceq printTransitionList({C:Conf, R:RuleName} TL:TransitionList) =
    printConf(C:Conf) '\b '-> '\o printTransitionList(TL:TransitionList)
 if TL:TransitionList =/= nil .

 op printModelCheckResult : ModelCheckResult QidList -> QidList .
 eq printModelCheckResult(B:Bool, QL:QidList) =
    'IMP: '\b 'Model 'check 'result 'to 'command
    '\c QL:QidList '\o '\b ': '\o '\n
        metaPrettyPrint(upModule('BOOL, false), upTerm(B:Bool)) .

 eq printModelCheckResult(
    counterexample(TL1:TransitionList, TL2:TransitionList), QL:QidList) =
    'IMP: '\b 'Model 'check 'counter 'example 'to 'command
    '\c QL:QidList '\o '\b ': '\o '\n
    '\b 'Path 'from 'the 'initial 'state: '\o
    '\n printTransitionList(TL1:TransitionList)
    '\n '\b 'Loop: '\o
    '\n printTransitionList(TL2:TransitionList) .
endm
```

6.5.4 IMP command line interface

⟨*imp-interface*⟩≡

```
mod IMP-INTERFACE is
 pr LOOP-MODE * (sort State to LoopState).
 pr COMPILE-IMP-TO-BPLC .
 pr IMP-PRETTY-PRINTING .

 sorts Dec? MetaIMPModule Command IMPState .
 subsort Term < MetaIMPModule .
 subsort Dec < Dec? .
 subsort IMPState < LoopState .

 op noDec : -> Dec? .
 op noModule : -> MetaIMPModule .
 op idle : -> Command .
 op <_;_;_> : MetaIMPModule Dec? QidList -> IMPState .
 op init : -> System .
 op init‘IMP‘no‘banner : -> System .
 op banner : -> QidList .

 eq banner = ’\g ’IMP ’Prototype ’\o ’
             ’‘( ’\y ’\! ’March ’2018 ’\o ’‘) .
 eq init = [nil, < noModule ; noDec ; nil >, banner] .
 eq init‘IMP‘no‘banner = [nil, < noModule ; noDec ; nil >, nil] .

 vars QIL QIL’ QIL’’ QIL1 QIL2 : QidList .

 rl [version] : [’version, L:LoopState, QIL] =>
                [nil, L:LoopState, banner] .

 *** Loading a module.
 crl [in] : [’module Q:Qid QIL,
            < M:MetaIMPModule ; D:Dec? ; QIL’ >, QIL’’] =>
  if (T:ResultPair? :: ResultPair)
  then [nil, < getTerm(T:ResultPair?) ;
               compileMod(getTerm(T:ResultPair?)) ;
               ’IMP: ’\b ’Module Q:Qid ’loaded. ’\o  >, QIL’’]
  else [nil, < noModule ; noDec ; nil >,
        printParseError(’module Q:Qid QIL, T:ResultPair?)]
  fi
 if T:ResultPair? :=
    metaParse(upModule(’IMP-GRAMMAR, false),
     ’module Q:Qid QIL, ’ModuleDecl) .

 *** Viewing a module.
 crl [view] : [’view , < M:MetaIMPModule ; D:Dec ; QIL’ >, QIL’’] =>
              [nil, < M:MetaIMPModule ; D:Dec ; QIL’ >,
                          printModule(M:MetaIMPModule)]
```

```
 if M:MetaIMPModule =/= noModule .

*** Viewing a module.
crl [python] : ['pie , < M:MetaIMPModule ; D:Dec ; QIL' >, QIL''] =>
               [nil, < M:MetaIMPModule ; D:Dec ; QIL' >,
                     dec2Decl(D:Dec, D:Dec, 0)]
 if M:MetaIMPModule =/= noModule .

*** Executing a command.
crl [exec-unbounded] : [('exec QIL),
                        < M:MetaIMPModule ; D:Dec ; QIL' >, QIL''] =>
 if P:ResultPair? :: ResultPair
 then [nil, < M:MetaIMPModule ; D:Dec ; QIL' > ,
       if compileCmd(getTerm(P:ResultPair?)) :: Cmd
       then
         'IMP: '\b 'Execution 'result 'for '\c QIL '\o '\b ': '\o
         printExec(getTerm(metaRewrite(upModule('BPLC+META-LEVEL, false),
          'run[upTerm(compileCmd(getTerm(P:ResultPair?))),
               upTerm(D:Dec)], unbounded)))
       else 'IMP: '\r 'Internal 'Error 'while 'compiling '\s QIL
       fi ]
 else [nil, < M:MetaIMPModule ; D:Dec ; QIL' >,
       'IMP: '\r 'Error 'while 'processing 'command '\o 'exec '\s QIL ]
 fi
 if P:ResultPair? :=
    metaParse(upModule('IMP-GRAMMAR, false), QIL, 'CommandDecl) .

*** Model check command
crl [mc] : [('mc QIL1 '|= QIL2),
            < 'module__end[T1:Term, T2:Term] ; D:Dec ; QIL' >, QIL''] =>
 if ((P1:ResultPair? :: ResultPair) and (P2:ResultPair? :: ResultPair))
 then [nil, < 'module__end[T1:Term, T2:Term] ; D:Dec ; QIL' >,
       if compileCmd(getTerm(P1:ResultPair?)) :: Cmd
       then
        printModelCheckResult(downTerm(
         getTerm(metaReduce(
           makeMCModule(printTokens(T1:Term), D:Dec),
            '_`,_|=?_[upTerm(D:Dec),
           upTerm(compileCmd(getTerm(P1:ResultPair?))),
            getTerm(P2:ResultPair?)])),true), QIL1 '\s '|= '\s QIL2)
       else 'IMP: '\r 'Internal 'Error 'while
            'processing 'command '\o 'mc QIL1 '|= QIL2
       fi ]
 else [nil, < 'module__end[T1:Term, T2:Term] ; D:Dec ; QIL' >,
              'IMP: '\r 'Error 'while 'processing 'command '\o
              'mc QIL1 '|= QIL2]
 fi
if P1:ResultPair? :=
      metaParse(upModule('IMP-GRAMMAR, false), QIL1, 'CommandDecl) /\
```

```
      P2:ResultPair? :=
         metaParse(makeMCModule(printTokens(T1:Term), D:Dec), QIL2, 'Formula) .

   *** Output
   crl [out] : [nil, < M:MetaIMPModule ; D:Dec? ; QIL' > , QIL''] =>
               [nil, < M:MetaIMPModule ; D:Dec? ; nil > , QIL']
    if QIL' =/= nil .

   *** Command error
   crl [com-error] : [Q:Qid QIL, < M:MetaIMPModule ; D:Dec ; QIL' > , QIL''] =>
       [nil, < M:MetaIMPModule ; D:Dec ; QIL' > ,
        'IMP: '\r 'Error 'no 'such 'command '\o Q:Qid QIL]
    if (Q:Qid =/= 'module) /\ (Q:Qid =/= 'view) /\
       (Q:Qid =/= 'exec) /\ (Q:Qid =/= 'mc) /\ (Q:Qid =/= 'pie) .
  endm

  loop init .
```

⟨*imp-grammar*⟩≡
  ⟨*token-module*⟩
  ⟨*imp-grammar-modules*⟩

⟨*imp-grammar-modules*⟩≡
  ⟨*predicate-decl-module*⟩
  ⟨*command-decl-module*⟩
  ⟨*imp-grammar-module*⟩

⟨*predicate-decl-module*⟩≡

```
  ***(Internal: Constants are typed at the metalevel:
  op c : -> S becomes the metaterm 'c.S.
  To avoid including IMP-GRAMMAR into COMPILE-IMP-TO-BPLC-EXP, we
  "cast out", from IMP-GRAMMAR, PREDICATE-DECL and COMMAND-DECL
  sorts and constants.)

  fmod PREDICATE-DECL is
   sort PredicateDecl .
   ops true false : -> PredicateDecl .
  endfm
```

⟨*command-decl-module*⟩≡

```
  fmod COMMAND-DECL is
   sort CommandDecl .
   op nop : -> CommandDecl .
  endfm
```

⟨*compiler-to-bplc*⟩≡
  ⟨*load-bplc*⟩
  ⟨*compile-imp-to-bplc*⟩

⟨*load-bplc*⟩≡

```
  load ../../../maude/bplc.maude
```

⟨*Compiling IMP expressions to BPLC Exp*⟩≡
⟨*Compiling tokens*⟩
⟨*Compiling arithmetic expressions*⟩
⟨*Compiling boolean expressions*⟩

⟨*imp.maude*⟩≡
⟨*imp-grammar*⟩
⟨*compiler-to-bplc*⟩
⟨*imp-pretty-printing*⟩
⟨*imp-interface*⟩

Chapter 7

A π compiler for the Abstract Machine Notation in Maude

The B method [1] is an outstanding formal method for safety critical systems, in particular, railway systems. Such systems have quite strong requirements such as real-time constraints and fault (in)tolerance due to the nature of their problem domain that may involve lives. Therefore, they are natural candidates for the application of formal methods for their specification and validation.

One of the main components of the B method are machine descriptions in the Abstract Machine Notation (AMN). They describe how a software component must behave in a safety-critical system. Such descriptions are suitable for an automata-based interpretation and therefore can be naturally specified as transition systems [14] and validated by techniques such as model checking [14].

This chapter proposes B Maude, a tool for the validation of AMN descriptions. Our approach applies formal semantics of programming languages techniques. It is comprised by a set of basic programming languages constructs, inspired on Peter Mosses' Component-based Semantics [27] whose semantics are given in terms of a generalization of Gordon Plotkin's Interpreting Automata [30]. We named the resulting formalism π Framework [6]. The semantics of AMN statements is given in terms of π constructions, or π lib as we call it. B Maude is the implementation of the π Framework and the transformation from AMN to π in the Rewriting Logic language Maude [16]. It allows for execution by rewriting, symbolic search with narrowing and Linear Temporal Logic model checking of AMN descriptions. The B Maude prototype can be downloaded from `https://github.com/ChristianoBraga/BMaude`.

This chapter is organized as follows. In Section 7.1 we recall the syntax of AMN and the use of Maude as a formal meta-tool. Section 7.2 formalizes AMN in π by giving denotations of AMN statements as π lib constructions. Section 7.3 talks about B Maude by showing the application of search and model checking to an AMN description and outlines their implementation in B Maude. Section 7.4 discusses work related to the formal semantics of AMN. Section 9 wraps up this paper with a summary of its contents and points to future work.

7.1 Preliminaries

7.1.1 Abstract Machine Notation grammar and example

In B Maude, we organize the Abstract Machine Notation (AMN) syntax in two parts. The first one is a context-free grammar for AMN expressions and commands, called Generalized Substitution Lan-

guage. The second part declares machine level statements such as machines, variables, constants, values and operations. The AMN grammar will be used in Section 7.2 when we specify the semantics of AMN as denotations in the π Framework. Only an excerpt is covered just enough to describe the example in Listing 11.

The Generalized Substitution Language, which essentially defines *expressions* and *commands*, is comprised by:

- identifiers, denoted by non-terminal ⟨*GSLIdentifiers*⟩[1],
- arithmetic expressions, denoted by non-terminal ⟨*GSLExpression*⟩, predicates, denoted by non-terminal ⟨*GSLPredicate*⟩, and
- substitutions, denoted by non-terminal ⟨*GSLSubstitution*⟩.

The notation for expressions and predicates is quite standard. Substitutions are essentially commands in AMN. Command '`_:=_`' denotes an assignment (an infix operator where '`_`' denotes the positions of its operands), '`IF_THEN_END`' and '`IF_THEN_ELSE_END`' denote conditionals, and '`WHILE_DO`' denotes unbounded repetition. Operator '`_OR_`' represents the bounded choice substitution. Keyword '`BEGIN_END`' is a declaration and yields a scope where its substitutions must be evaluated within. The GSL grammar (excerpt) in BNF notation is as follows.

⟨*GSLIdentifiers*⟩ ::= '`bid(`'⟨*ID*⟩'`)`' | '`grat(`' ⟨*RAT*⟩'`)`' | '`gboo(`' ⟨*BOOL*⟩'`)`'

⟨*GSLExpression*⟩ ::= ⟨*GSLIdentifiers*⟩ | ⟨*GSLPredicate*⟩
| ⟨*GSLExpression*⟩ '`+`' ⟨*GSLExpression*⟩ | ...

⟨*GSLPredicate*⟩ ::= ⟨*GSLExpression*⟩ '`==`' ⟨*GSLExpression*⟩
| ⟨*GSLExpression*⟩ '`/\`' ⟨*GSLExpression*⟩ | ...

⟨*GSLSubstitution*⟩ ::= ⟨*GSLIdentifiers*⟩ := ⟨*GSLExpression*⟩
| '`IF`' ⟨*GSLPredicate*⟩ '`THEN`' ⟨*GSLSubstitution*⟩
'`ELSE`' ⟨*GSLSubstitution*⟩ '`END`'
| '`WHILE`' ⟨*GSLPredicate*⟩ '`DO`' ⟨*GSLSubstitution*⟩
| ⟨*GSLSubstitution*⟩ '`OR`' ⟨*GSLSubstitution*⟩
| '`BEGIN`' ⟨*GSLSubstitution*⟩ '`END`' ⟨*GSLSubstitution*⟩ | ...

The context-free grammar for AMN's machine-related statements essentially defines *declarations*:

- a '`MACHINE__END`' declaration, denoted by non-terminal ⟨*AMNMachine*⟩, declares variables, constants, initializations for either or both, and operations,
- a '`VARIABLES`' declaration (resp. '`CONSTANTS`'), in ⟨*AMNAbsVariables*⟩, is only a list of identifiers, and
- a ⟨*AMNValuation*⟩ declaration associates a substitution to the initialization of a variable or constant. An operation declaration, denoted by non-terminal ⟨*AMNOperation*⟩, associates a substitution to an (operation) identifier and a list of (identifier) parameters.

The AMN grammar in BNF notation is as follows.

⟨*AMNMachine*⟩ ::= '`MACHINE`' ⟨*GSLIdentifiers*⟩ ⟨*AMNClauses*⟩ '`END`' | '`MACHINE`' ⟨*GSLIdentifiers*⟩ '`END`'

[1] Non-terminals ⟨*ID*⟩, ⟨*RAT*⟩ and ⟨*BOOL*⟩ are left unspecified but simply denote what their names imply.

⟨*AMNClauses*⟩ ::= ⟨*AMNAbsVariables*⟩ | ⟨*AMNAbsConstants*⟩ | ⟨*AMNOperations*⟩ |
⟨*AMNValuesClause*⟩ | ⟨*AMNClauses*⟩ ⟨*AMNClauses*⟩

⟨*AMNAbsVariables*⟩ ::= 'VARIABLES' ⟨*AMNIdList*⟩

⟨*AMNAbsConstants*⟩ ::= 'CONSTANTS' ⟨*AMNIdList*⟩

⟨*AMNIdList*⟩ ::= ⟨*AMNIdList*⟩ ',' ⟨*AMNIdList*⟩

⟨*AMNValuesClause*⟩ ::= 'VALUES' ⟨*AMNValSet*⟩

⟨*AMNValuation*⟩ ::= ⟨*GSLIdentifiers*⟩ '=' ⟨*GSLExpression*⟩

⟨*AMNValSet*⟩ ::= ⟨*AMNValSet*⟩ ';' ⟨*AMNValSet*⟩

⟨*AMNOperations*⟩ ::= 'OPERATIONS' ⟨*AMNOpSet*⟩

⟨*AMNOperation*⟩ ::= ⟨*GSLIdentifiers*⟩ '=' ⟨*GSLSubstitution*⟩

⟨*AMNOpSet*⟩ ::= ⟨*AMNOpSet*⟩ ';' ⟨*AMNOpSet*⟩

⟨*AMNOperation*⟩ ::= ⟨*GSLIdentifiers*⟩ '(' ⟨*AMNIdList*⟩ ')' ⟨*GSLSubstitution*⟩

7.1.2 The MUTEX machine.

As an illustrative example, Listing 11 declares the MUTEX machine (which is actually executable in B Maude). It will be our running example in this paper. In Section 7.3, we will use it to explain how B Maude is implemented. The MUTEX machine specifies a simple mutual exclusion protocol with two processes competing to enter a critical section. Each process, represented by an abstract variable in the MUTEX machine, can be in one of two possible states: idle, wait, or crit, encoded as constants in the machine. The protocol is encoded as the (parameterless) operation mutex that simply runs forever and, depending on the state of the processes, may non-deterministically change one of the processes state. For example, substitution

```
1 IF p1 == idle /\ p2 == idle THEN p1 := wait OR p2 := wait ELSE ...
```

declares that, in a situation where both p1 and p2 are in the idle state, either one or both of them may change to wait.

```
1  MACHINE MUTEX
2    VARIABLES p1 , p2
3    CONSTANTS idle , wait , crit
4    VALUES
5      p1 = 0 ; p2 = 0 ;
6      idle = 0 ; wait = 1 ; crit = 2
7    OPERATIONS
8      mutex =
9        WHILE true DO
10       BEGIN
11         IF p1 == idle /\ p2 == idle
12         THEN (p1 := wait OR p2 := wait)
13         ELSE IF p1 == idle /\ p2 == wait
14         THEN p1 := wait OR p2 := crit
15         ELSE IF p1 == idle /\ p2 == crit
16         THEN p1 := wait OR p2 := idle
17         ELSE IF p1 == wait /\ p2 == idle
18         THEN p1 := crit OR p2 := wait
19         ELSE IF p1 == wait /\ p2 == wait
20         THEN p1 := crit OR p2 := crit
```

```
        ELSE IF p1 == wait /\ p2 == crit
        THEN p2 := idle
        ELSE IF p1 == crit /\ p2 == idle
        THEN p1 := idle OR p2 := wait
        ELSE IF p1 == crit /\ p2 == wait
        THEN p1 := idle
        END END END END END END END END
    END
END
```

Listing 11: MUTEX protocol in the Abstract Machine Notation

7.2 π denotations for the Abstract Machine Notation

This section formalizes the meaning of B's Abstract Machine Notation (AMN) in terms of π denotations. The subset of the AMN considered in this section is (almost) in bijection with π lib, so the formalization is quite easy to follow.

The π denotational semantics of AMN, denoted by function $[\![\cdot]\!]_\pi$, simply maps AMN statements, that is, arithmetic expressions, predicates, substitutions, machine declarations, variable declarations, constant declarations, initialization declarations, and operation declarations, into π denotations in a quite direct way.

GSL expressions, such as sum and conjunction, are mapped to π lib expressions, *add* and *and* in Equations 7.4 and 7.6. GSL substitutions are mapped to commands, such as '_:=_ being denoted by assign in Equation 7.7.

A 'MACHINE__END' declaration is associated with a *dec* construction in π lib. Such a declaration is a composition of further declarations for variables and constants, initialized according with the expressions in the 'VALUES' clause, and operations. Variable declarations give rise to *ref* declarations, that associate a location in the memory with a given identifier in the environment. (Constants are denoted by *cns* π lib declarations.) An operation declaration gives rise to a *prc* declaration that binds an abstraction (a list of formal parameters together with a block of commands) to an identifier in the environment. An operation call substitution is translated into a *cal* π lib command, parametrized by the expressions resulting from the translations of the GSLExpressions given as actual parameters in the given operation call.

7.3 B Maude

B Maude is a formal executable environment for the Abstract Machine Notation that implements, in the Maude language, the π denotational semantics described in Section 7.2. The main objective of B Maude is to endow the B method with the validation techniques available in the Maude system, both the buit-in ones, such as rewriting, narrowing, rewriting modulo SMT and model checking, and user-defined ones.

The current version of B Maude is called URUÇÚ AMARELA (a bee common in the Rio de Janeiro area), it requires Maude Alpha 115 or later to run due to some of the narrowing-based techniques available in this version. (Even though it is not directly available from Maude's web site, Alpha 115 can be obtained free of charge by joining the Maude (Ab)users group by sending an email to maude-users@cs.uiuc.edu.) Figure 7.1 displays B Maude banner and the output of correctly loading the MUTEX machine. (The logo makes a "visual pun" with hexagons from a bee hive and components, the sound of the B method's name and its objective of giving support to component-based development.)

B Maude's architecture is comprised essentially by the following components (recall the discussion in Section 2.2.2), together with the Maude implementation of the π Framework (see Section **??**):

Let F, I, P, V in ⟨*GSLIdentifiers*⟩, R in ⟨*RAT*⟩, B in ⟨*BOOL*⟩, and E, $\mathtt{E}_\mathtt{i}$ in ⟨*GSLExpression*⟩, P in ⟨*GSLPredicate*⟩, S, $\mathtt{S}_\mathtt{i}$ in ⟨*GSLSubstitution*⟩, A,$\mathtt{A}_\mathtt{i}$ in ⟨*GSLActuals*⟩, FS in ⟨*AMNIdList*⟩, O in ⟨*AMNOperation*⟩, OS in ⟨*AMNOpSet*⟩, VS in ⟨*AMNValSet*⟩, VC in ⟨*AMNValuesClause*⟩, AV in ⟨*AMNAbsVariables*⟩, in

$$[\![\mathtt{I}]\!]_\pi = id(\mathtt{I}), \tag{7.1}$$
$$[\![\mathtt{grat(R)}]\!]_\pi = rat(\mathtt{R}), \tag{7.2}$$
$$[\![\mathtt{gboo(B)}]\!]_\pi = boo(\mathtt{B}), \tag{7.3}$$
$$[\![\mathtt{E}_1 \mathtt{+} \mathtt{E}_2]\!]_\pi = add([\![\mathtt{E}_1]\!]_\pi, [\![\mathtt{E}_2]\!]_\pi), \tag{7.4}$$
$$[\![\mathtt{E}_1 \mathtt{==} \mathtt{E}_2]\!]_\pi = eq([\![\mathtt{E}_1]\!]_\pi, [\![\mathtt{E}_2]\!]_\pi), \tag{7.5}$$
$$[\![\mathtt{E}_1 \mathtt{/\backslash} \mathtt{E}_2]\!]_\pi = and([\![\mathtt{E}_1]\!]_\pi, [\![\mathtt{E}_2]\!]_\pi), \tag{7.6}$$
$$[\![\mathtt{I} \mathtt{:=} \mathtt{E}]\!]_\pi = assign([\![\mathtt{I}]\!]_\pi, [\![\mathtt{E}]\!]_\pi), \tag{7.7}$$
$$[\![\mathtt{S}_1 \mathtt{\ OR\ } \mathtt{S}_2]\!]_\pi = choice([\![\mathtt{S}_1]\!]_\pi, [\![\mathtt{S}_2]\!]_\pi), \tag{7.8}$$
$$[\![\mathtt{IF\ P\ THEN\ S}_1 \mathtt{\ ELSE\ S}_2]\!]_\pi = if([\![\mathtt{P}]\!]_\pi, [\![\mathtt{S}_1]\!]_\pi, [\![\mathtt{S}_2]\!]_\pi), \tag{7.9}$$
$$[\![\mathtt{WHILE\ P\ DO\ S}]\!]_\pi = loop([\![\mathtt{P}]\!]_\pi, [\![\mathtt{S}]\!]_\pi) \tag{7.10}$$
$$[\![\mathtt{MACHINE\ I\ AV\ C\ VC\ OP\ END}]\!]_\pi = dec([\![\mathtt{AV,VC}]\!]_\pi, dec([\![\mathtt{C,VC}]\!]_\pi, [\![\mathtt{OP}]\!]_\pi)), \tag{7.11}$$
$$[\![\mathtt{MACHINE\ I\ AV\ VC\ OP\ END}]\!]_\pi = dec([\![\mathtt{AV,VC}]\!]_\pi, [\![\mathtt{OP}]\!]_\pi), \tag{7.12}$$
$$[\![\mathtt{MACHINE\ I\ C\ VC\ OP\ END}]\!]_\pi = dec([\![\mathtt{C,VC}]\!]_\pi, [\![\mathtt{OP}]\!]_\pi), \tag{7.13}$$
$$[\![\mathtt{MACHINE\ I\ OP\ END}]\!]_\pi = [\![\mathtt{OP}]\!]_\pi, \tag{7.14}$$
$$[\![\mathtt{VARIABLES\ IS,VALUES\ VS}]\!]_\pi = [\![\mathtt{IS,VS}]\!]_\pi, \tag{7.15}$$
$$[\![\mathtt{I,I=E}]\!]_\pi = ref([\![\mathtt{I}]\!]_\pi, [\![\mathtt{E}]\!]_\pi), \tag{7.16}$$
$$[\![\mathtt{I,(I=E;VS)}]\!]_\pi = ref([\![\mathtt{I}]\!]_\pi, [\![\mathtt{E}]\!]_\pi), \tag{7.17}$$
$$[\![\mathtt{(I\,,IS),(I=E;VS)}]\!]_\pi = dec(ref([\![\mathtt{I}]\!]_\pi, [\![\mathtt{E}]\!]_\pi), [\![\mathtt{IS,VS}]\!]_\pi), \tag{7.18}$$
$$[\![\mathtt{I},(\mathtt{I'=E;VS})]\!]_\pi = [\![\mathtt{I,VS}]\!]_\pi, if\ \mathtt{I} \neq \mathtt{I'} \tag{7.19}$$
$$[\![\mathtt{(I\,,IS),(I'=E;VS)}]\!]_\pi = [\![\mathtt{(I\,,IS),VS}]\!]_\pi, if\ \mathtt{I} \neq \mathtt{I'} \tag{7.20}$$
$$[\![\mathtt{OPERATIONS\ O}]\!]_\pi = [\![\mathtt{O}]\!]_\pi, \tag{7.21}$$
$$[\![\mathtt{OPERATIONS\ OS}]\!]_\pi = [\![\mathtt{OS}]\!]_\pi, \tag{7.22}$$
$$[\![\mathtt{O;OS}]\!]_\pi = dec([\![\mathtt{O}]\!]_\pi, [\![\mathtt{OS}]\!]_\pi), \tag{7.23}$$
$$[\![\mathtt{P=S}]\!]_\pi = prc([\![\mathtt{P}]\!]_\pi, blk([\![\mathtt{S}]\!]_\pi)), \tag{7.24}$$
$$[\![\mathtt{P(FS)=S}]\!]_\pi = prc([\![\mathtt{P}]\!]_\pi, [\![\mathtt{FS}]\!]_\pi^{for}, blk([\![\mathtt{S}]\!]_\pi)), \tag{7.25}$$
$$[\![\mathtt{F}]\!]_\pi^{for} = par([\![\mathtt{F}]\!]_\pi^{for}), \tag{7.26}$$
$$[\![\mathtt{F\,,FS}]\!]_\pi^{for} = for([\![\mathtt{F}]\!]_\pi^{for}, [\![\mathtt{FS}]\!]_\pi^{for}), \tag{7.27}$$
$$[\![\mathtt{I()}]\!]_\pi = cal([\![\mathtt{I}]\!]_\pi), \tag{7.28}$$
$$[\![\mathtt{I(E)}]\!]_\pi = cal([\![\mathtt{I}]\!]_\pi, [\![\mathtt{E}]\!]_\pi), \tag{7.29}$$
$$[\![\mathtt{I(A)}]\!]_\pi = cal([\![\mathtt{I}]\!]_\pi, [\![\mathtt{A}]\!]_\pi^{act}), \tag{7.30}$$
$$[\![\mathtt{E}]\!]_\pi^{act} = [\![\mathtt{E}]\!]_\pi, \tag{7.31}$$
$$[\![\mathtt{E\,,A}]\!]_\pi^{act} = act([\![\mathtt{E}]\!]_\pi, [\![\mathtt{A}]\!]_\pi^{act}). \tag{7.32}$$

π Denotation 7.1: Abstract Machine Notation

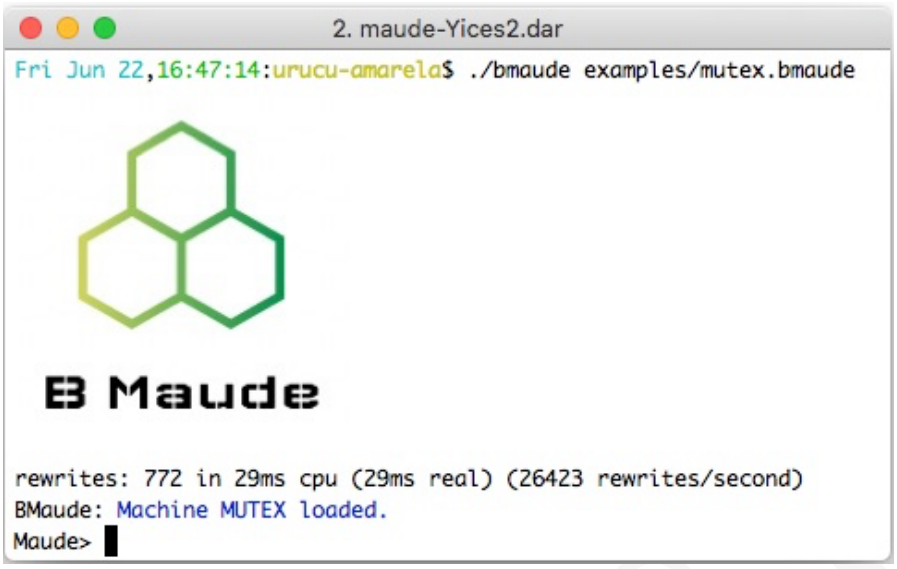

Figure 7.1: B Maude banner and loading a machine

(i) B Maude *parser* that, given a list of (quoted) identifiers, produces a term in the initial algebra of the module `AMN-SYNTAX`, representing B Maude's grammar; (ii) B Maude to π lib *compiler*, that is exactly the implementation of the π denotations of Section 7.2 in Maude; (iii) B Maude *read-eval-loop*, a user interface that processes commands, given in terms of quoted identifiers, calls the appropriate meta-function; (iv) B Maude *pretty-printer*, that translates the results coming from the meta-functions to user level representation, such as the output of the model checker in terms of π lib constructions, into B Maude syntax. In the following sections we describe the implementation of some of the functionality implemented in the tool: (i) searching the state space, in Section 7.3.1, and (ii) model checking machines, in Section 7.3.2.

7.3.1 Searching the computation graph of a GSL substitution

One may animate a B Maude description or check for invariants using the `search` command. Figure 7.2 displays the execution of a `search` command querying for the first solution that satisfies the constraint `p1 = 2`, denoting the situation where process 1 is in the critical section.

The `search` command is handled in two levels. First, the syntax of the command is checked at module `BMAUDE-INTERFACE`, essentially to make sure that the conditions are well-formed. Then, if that is the case, rule `search`, in module `BMAUDE-COMMANDS`, is applied. First the descent function `metaSearch` is invoked by `bMaudeSearch`, with appropriate parameters. If `metaSearch` succeeds, then `bMaudePrintTrace` pretty-prints the trace, from the *initial state*, given by the π automaton state

```
1 < cnt : blk(compile(M:AMNMachine), compile(S:GSLSubstitution)) ecs,
2   env : noEnv, sto : noStore , val : evs ,
3   locs : noLocs , out : evs , exc : CNT >,
```

Listing 12: Initial state for `metaSearch`

to the `N:Nat`-th solution, by invoking the descent function `metaSearchPath`. The initial state is such that the control stack `cnt` has a block resulting from the commands yielded by compilation (or π denotation) of the given GSL substitution `S:GSLSubstitution` (a call to operation `mutex`, in Figure 7.2)

together with the declarations resulting from the compilation of the machine `M:AMNMachine` (machine `MUTEX`, in this example) and remaining semantic components initialized to their default initial values. (For instance, `evs` is short for empty-value-stack and `CNT`, short for continue, means "normal execution", as opposed to an `EXT`, short for exit, meaning abnormal termination, that π lib construction *exit* may raise.)

```
crl [search] :
    < search N:Nat S:GSLSubstitution C:Condition ; M:AMNMachine ; QIL > =>
    < idle ; M:AMNMachine ;
        if T:ResultTriple? :: ResultTriple
        then
            bMaudePrintTrace(S:GSLSubstitution, M:AMNMachine, N:Nat)
        else
            printNoSolutionOrError(S:GSLSubstitution)
        fi >
if T:ResultTriple? := bMaudeSearch(S:GSLSubstitution, M:AMNMachine, N:Nat)
```

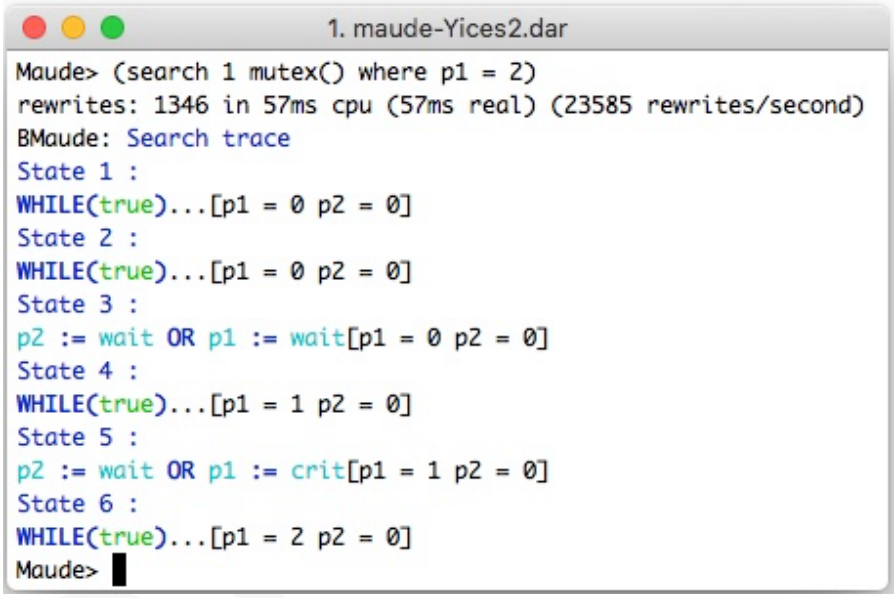

Figure 7.2: Mutex simulation in B Maude

Figure 7.3 is an example of the use of the `search` command to check for an *invariant* property. In this example, a *safety* property is checked by querying for a state such that both processes are in the critical section (that is, `p1 = 2 and p2 = 2`). Since no solution is found, then the protocol implemented by operation `mutex` is safe.

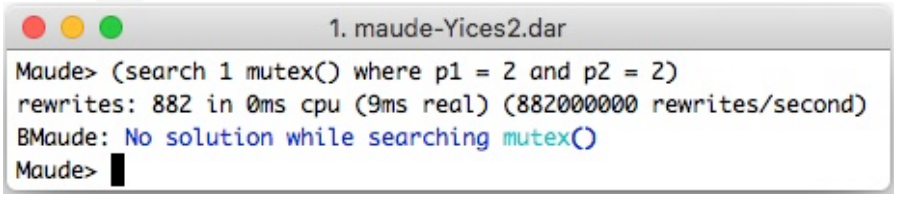

Figure 7.3: Checking mutex safety invariant with search in B Maude

7.3.2 Linear Temporal Logic model checking AMN Machines

In Section 3.4 we discussed how to model check π automata and in Section 7.2 we gave denotations for Abstract Machine Notation (AMN) statements as π lib constructions. Their combination yields the foundation to model check AMN descriptions for Linear Temporal Logic (LTL) properties. In B Maude, this is encoded as a meta-function that invokes the Maude LTL model checker to validate

$$\mathscr{K}([\![A]\!]_\pi), s_0 \models \varphi,$$

where $\mathscr{K}([\![A]\!]_\pi)$ is the Kripke structure associated with the π denotations for AMN description A, s_0 is defined as in Listing 12, and φ is an LTL formula such that the atomic propositions are instances of the Equation Schema **??**, as defined in Section 3.4.

As with the `search` command, the model check command is handled in two levels. The first level makes sure that the command is well-formed. Then, Rule `mc` is applied and actually invokes the model checker by calling `metaReduce` with: (i) the module resulting from the application of operation `makeMCModule`, that creates a meta-module that includes Maude's MODEL-CHECKER module and declares state propositions resulting from the variable declarations in A and, (ii) a meta-term denoting an invocation of the `modelCheck` operation (in module MODEL-CHECKER) that model checks $\mathscr{K}([\![A]\!]_\pi)$, starting in the π state that has the π denotation of `S:GSLSubstitution` on top of the control stack, as in Listing 12, for the LTL properties encoded in `T:Term`.

```
crl [mc] :
    < mc S:GSLSubstitution T:Term ; M:AMNMachine ; QIL > =>
    < idle ; M:AMNMachine ;
            if (properResultPair?(T:ResultPair?))
            then printModelCheckResult(T:ResultPair?))
            else printModelCheckError(S:GSLSubstitution, T:Term)
            fi >
if T:ResultPair? :=
   metaReduce(makeMCModule(M:AMNMachine),
    '_`,_|=?_[upTerm(M:AMNMachine), upTerm(S:GSLSubstitution), T:Term]) .
```

In Figure 7.4 we see that machine `MUTEX` does not have the liveness property, that specifies that if a process tries to enter the critical section it will eventually do so, by showing a counter-example where the system enters a loop with only one of the processes, p_2 in this scenario, accessing the critical section.

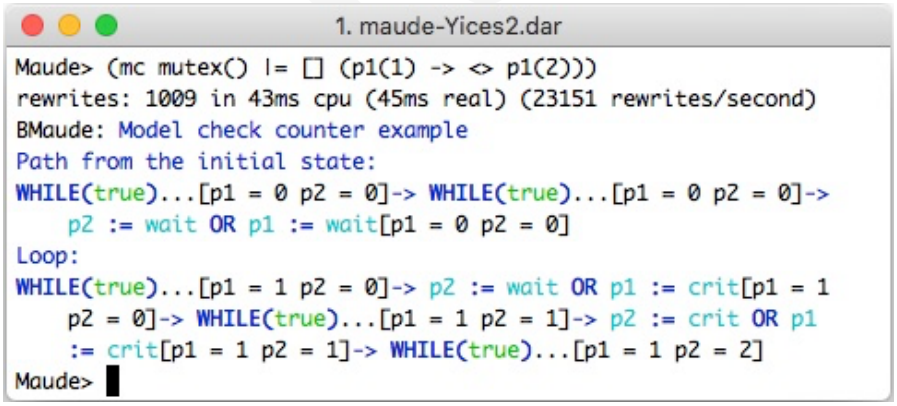

Figure 7.4: Checking mutex liveness in B Maude

7.4 Related work

In [7] the present authors together with David Deharbe and Anamaria Moreira proposed a Structural Operational Semantics for the Generalized Substitution Language (GSL). The SOS semantics for GSL had a straight forward representation in the Maude language by encoding SOS transition rules as conditional rules in Maude. Moreover, the Conditional Rewriting Logic Semantics of GSL in Maude was shown to have an equivalent Unconditional Rewriting Logic Semantics by considering conditional rewrites in the main rewrite "thread". The inspiration for the approach used there is the same here: Plotkin's Interpreting Automata. We have further developed this approach for operational semantics that is now called π Framework. The semantics of the Abstract Machine Notation is then described as denotations in the π Framework, as described in Section 7.2.

GSL is presented in [1] with a weakest precondition (WP) semantics. Most of the work on B and GSL relies on this semantics. An interesting example is the work of Dunne in [18], which includes additional information concerning the variables in scope at a substitution to solve some delicate issues that restrict what can be stated in B due to limitations of the pure WP semantics. On the other hand, some related work proposing the embedding of B into other formalisms with strong tool support, such as Isabelle/HOL, can be found in the literature [12,17]. As in the current research, the purpose of those works is combining strengths of both worlds, to achieve further proof or animation goals. On a more theoretical line, but also with similar goals, the work in [34] proposes a prospective value semantics to define the effect of GSL substitutions on values and expressions. The meaning of a computation, specified in GSL, is given in terms of the values of an expression, if the computation is carried out. In the current paper we contribute to this line of work by discussing GSL operational semantics, its rewriting logic semantics, using Maude as the specification language for Rewriting Logic theories, and a prototype execution environment in the Maude system.

Chapter 8

Related work

First and foremost there is the work by Peter Mosses on Component-Based Semantics [26] and funcons [13, 29], where programming language constructs are specified in Modular Structural Operational Semantics (MSOS). π lib is inspired by this research and is also a result of the research on the relation between MSOS and Rewriting Logic, with an implementation in Maude, that started in [8,25], with Edward Hermann Haeusler, Peter Mosses and José Meseguer, and continued with Fabricio Chalub [10, 11]. Despite their common roots, funcons and π lib have different models. The models of funcons are Arrow-labeled Transition Systems and π lib descriptions are to be interpreted as π Automata, as described in Section 3. π Automata can be understood as unlabeled transition systems. This makes it easy to relate π Automata with term rewriting systems and to have an efficient implementation of them when transition rules are mapped to unconditional rewrite rules. This is in contrast, for instance, with previous work by Chalub and the author in the MSOS Tool in Maude [11], that understands transition rules in MSOS as conditional rewrite rules in Maude.

In [29], Mosses and Vesely propose an implementation of Component-Based Semantics using the K Framework (e.g. [32]). K aims at being a methodology to define languages with tools for formal language development. It is based on concepts from Rewriting Logic Semantics, with some intuitions from Chemical Abstract Machines [5] (CHAMs) and Reduction Semantics [19] (RS). Abstract computational structures contain context needed to produce a future computation (like continuations). Computations take place in the context of a configuration, which are hierarchically made up of K *cells*. Each cell holds specific pieces of information such as computations, the environment, and memory store. K specifications allow for equations and rules. Equations (representing heating and cooling processes) manipulate term structure as opposed to rules that are computational and may be concurrent, similar to how Rewriting Logic understands equations and rules. K has stablished itself as a powerful framework for language semantics (e.g. the formal semantics for the Ethereum Virtual Machine [22]). However, it has a non-trivial model, with many different concepts, coming from different frameworks such as MSOS, Rewriting Logic, Reduction Semantics, and CHAM. The combination of Component-Based Semantics and K in [29] provides indeed a powerful tool for language semantics descriptions.

π Automata, as described in Section 3, is a less ambitious framework while compared with K, being conceived to be simple, easily integrated into an undergraduate level course, and with an efficient implementation in Maude, as K is. As a matter of fact, it has several intersections with K given their common roots in Rewriting Logic Semantics and MSOS. Due to π Automata' simpler automata-based model, it appears that it is a nicer candidate to teach formal semantics and compiler construction than K. It smoothly connects with Introduction to Programming Languages, Programming Languages Semantics and Formal Languages and Automata Theory, with good properties such as expressivity,

efficiency and support for automata-based automated specification and reasoning.

Chapter 9

Conclusion

Summary. This manuscript discusses the π framework for teaching formal compiler construction. It has a denotational character, and its implementation called π lib builds on Peter Mosses Component-Based Semantics [28]. The framework implements a library of common programming languages constructions, such as assignments, function declarations and function calls. The semantics of a programming language is then given in a syntax-directed way, by expressing the denotations of the given programming language constructs in terms of π lib elements. The semantics of π lib is also described formally. Each element in π lib is specified in terms of π Automata. Essentially, a π Automata describes both static and dynamic semantics by means of (unconditional) rules that relate sets of semantic components, such as the memory store, the environment, a control stack and a value stack. π Automata is overloaded to refer also to a π Automata with a set of state propositions that are used to validate a given π Automata using automata-based techniques such as model checking. π Automata is a generalization of Plotkin's Interpreting Automata [30]. Currently, the π approach is implemented in Maude yielding an effective tool for formal compiler construction and program verification. The latter is accomplished when the formal tools in Maude, such as term rewriting, narrowing and LTL model checking, are lifted to a given programming language in π. The current prototype implementation of π in Maude is available at `http://github.com/ChristianoBraga/BPLC`, with an implementation for an imperative language called IMP, available in the same repository.

Preliminary assessment and future work. π appears to be a suitable approach to teach compiler construction, as much as Component-Based Semantics is to teach formal semantics of programming languages [28] since one only works with a small set of programming constructions that may be used to give semantics to different programming languages, in different paradigms, with a model amenable to automated verification, as discussed in Section **??**. The approach proposed in this manuscript has been class tested for the past year, with quite positive results. All students have completed their projects within the academic semester, reporting it back as a rewarding experience, with a lot of work. The π framework appears to ease understanding of the meaning of the constructions when compared to their SOS counterparts. Even though a complete Maude implementation of π is available as reference, ways of stimulating its use and sandboxing with it need to be developed. In this context, perhaps an interesting discussion regards the definition of a meta-language for describing π compilers. At first, our intention is to make the π library available in different programming languages and let one choose one's preferred parsing/transformation framework. However, this choice appears to have some undesirable pedagogical consequences. The Maude implementation of π, for instance, uses meta-programming techniques that create some resistance to the understanding of the rather simple aspects of π lib and its π Automata semantics. The author foresees the continuation of this work by addressing the issues raised in this preliminary assessment and by extending the π lib library with new

constructs, improving code generation and validation techniques.

Acknowledgements Fabricio Chalub, Edward Hermann Hauesler, José Meseguer, and Peter D. Mosses for the long term collaboration that inspired the work discussed in this manuscript, and to Narciso Martí-Oliet for the many contributions as a co-author of B Maude.

Bibliography

[1] J.-R. Abrial. *The B-book - Assigning Programs to Meanings.* Cambridge Univ. Press, 2005.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd. edition, 2006.

[3] A. Arnold. *Finite Transition Systems: Semantics of Communicating Systems.* Prentice Hall, Upper Saddle River, New Jersey 07458, 1994.

[4] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, New York, NY, USA, 1998.

[5] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217 – 248, 1992.

[6] C. Braga. Compiler Construction with Basic Programming Languages Constructs and Generalized Interpreting Automata. *ArXiv e-prints*, May 2018.

[7] C. Braga, D. Deharbe, A. M. Moreira, and N. M. Oliet. A rewriting logic semantics for the generalized substitution language. In *Proceedings of School of Theoretical Computer Science and Formal Methods (ETMF 2016)*, ISBN 978-85-7669-357-4, pages 93–104. Sociedade Brasileira de Computação, 2016.

[8] C. Braga and J. Meseguer. Modular rewriting semantics in practice. In N. Martí-Oliet, editor, *Proceedings of 5th International Workshop on Rewriting Logic and its Applications, WRLA 2004*, volume 117, pages 393–416. Elsevier, 2005.

[9] F. Chalub. An implementation of Modular Structural Operational Semantics in Maude. Master's thesis, Universidade Federal Fluminense, 2005.

[10] F. Chalub and C. Braga. A modular rewriting semantics for CML. *Journal of Universal Computer Science*, 10(7):789–807, 2004.

[11] F. Chalub and C. Braga. Maude MSOS Tool. *Electronic Notes in Theoretical Computer Science*, 176(4):133–146, 2006.

[12] P. Chartier. Formalisation of B in Isabelle/HOL. In D. Bert, editor, *B'98: Recent Advances in the Development and Use of the B Method, Montpellier, France, April 22–24, 1998 Proceedings*, pages 66–82. Springer Berlin Heidelberg, 1998.

[13] M. Churchill, P. D. Mosses, N. Sculthorpe, and P. Torrini. Reusable components of semantic specifications. In S. Chiba, É. Tanter, E. Ernst, and R. Hirschfeld, editors, *Transactions on Aspect-Oriented Software Development XII*, pages 132–179, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[14] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[15] 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.

[16] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude*, volume 4350 of *LNCS*. Springer, 2007.

[17] D. Déharbe and S. Merz. Software component design with the B method - A Formalization in Isabelle/HOL. In C. Braga and C. P. Ölveczky, editors, *Formal Aspects of Component Software: 12th Inter. Conf., FACS 2015, Niterói, Brazil, October 14-16, 2015*, pages 31–47. Springer, 2016.

[18] S. Dunne. A theory of generalised substitutions. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002:Formal Specification and Development in Z and B: 2nd Int. Conf. of B and Z Users, Grenoble, France, January 23–25, 2002, Proc.*, pages 270–290. Springer, 2002.

[19] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235 – 271, 1992.

[20] J. A. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, Cambridge, MA, USA, 1996.

[21] 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.

[22] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, and G. Rosu. Kevm: A complete semantics of the ethereum virtual machine. Technical report, University of Illinois at Urbana Chanpaign, http://hdl.handle.net/2142/97207, 08 2017.

[23] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In *Handbook of Philosophical Logic*, volume 9, pages 1–87. Kluwer Academic Publishers, P.O. Box 17, 3300 AA Dordrecht, the Netherlands, 2002.

[24] J. Meseguer. Conditional rewriting as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, April 1992.

[25] J. Meseguer and C. Braga. Modular rewriting semantics of programming languages. In C. Rattray, S. Maharaj, and C. Shankland, editors, *In Algebraic Methodology and Software Technology: proceedings of the 10th International Conference, AMAST 2004*, volume 3116 of *LNCS*, pages 364–378, Stirling, Scotland, UK, July 2004. Springer.

[26] 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.

[27] P. D. Mosses. Component-based semantics. In *Proc. of the 8th Inter. Workshop on Spec. and Verif. of Comp.-based Systems (SAVCBS '09)*, pages 3–10. ACM, 2009.

[28] 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.

[29] P. D. Mosses and F. Vesely. Funkons: Component-based semantics in k. In S. Escobar, editor, *Rewriting Logic and Its Applications*, pages 213–229, Cham, 2014. Springer International Publishing.

[30] G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004. Special issue on SOS.

[31] 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.

[32] G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[33] P. Viry. Elimination of conditions. *J. Symb. Comput.*, 28(3):381–400, 1999.

[34] F. Zeyda, B. Stoddart, and S. Dunne. A prospective-value semantics for the GSL. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *ZB 2005: Formal Spec. and Dev. in Z and B: 4th Int. Conf. of B and Z Users, Guildford, UK, April 13-15. Proc.*, pages 187–202. Springer, 2005.