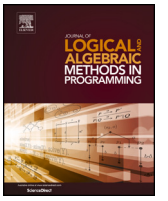
J. Log. Algebraic Methods Program. 142 (2025) 101005

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

journal homepage: www.elsevier.com/locate/jlamp



Maude2Lean: Theorem proving for Maude specifications using Lean

Rubén Rubio [a,*], Adrián Riesco [a,b]

[a] Facultad de Informática, Universidad Complutense de Madrid, Spain
[b] Instituto de Tecnología del Conocimiento, Universidad Complutense de Madrid, Spain

ARTICLE INFO

Keywords:
Theorem proving
Rewriting logic
Maude
Lean

ABSTRACT

Maude is a specification language based on rewriting logic whose programs can be executed, model checked, and analyzed with other automated techniques, but not easily theorem proved. On the other hand, Lean is a modern proof assistant based on the calculus of inductive constructions with a wide library of reusable proofs and definitions. This paper presents a translation from the first formalism to the second, and the `maude2lean` tool that predictably derives a Lean program from a Maude module. Hence, theorems can be proved in Lean about Maude specifications.

1. Introduction

Formal methods comprise a wide spectrum of specification formalisms and verification techniques for describing and checking the behavior of systems against their expected properties. These formalisms are usually influenced by the intended applications and verification approach, with varying automation and user intervention degrees. For instance, model checking has been widely used in industry because of the unattended nature and smooth learning curve, but it can only work for concrete size-bounded instances of a system unless non-trivial abstractions are used. On the contrary, interactive theorem proving demands more effort, knowledge, and training from the user, but it can handle more general properties by induction [33]. For effective verification of different kinds of properties in an affordable way, the possibility of applying multiple techniques on the same specification and interoperating with different tools is useful and convenient.

Maude [7,8] is a specification language based on rewriting logic [26,27], which extends membership equational logic [5] with non-deterministic rewrite rules that naturally represent concurrent computation. For modeling a system under this formalism, states are described as terms in an order-sorted signature modulo equations, while transitions are represented by rewrite rules acting on these terms. Specifications in Maude are organized in modules and can be executed with the multiple commands available in the interpreter. This combination of a simple but expressive syntax, powerful semantics, and efficient implementation makes Maude an appropriate tool for specifying a large number of systems [27,25]. Moreover, a builtin LTL model checker [19] and other verification tools turn Maude into a convenient tool for formal verification. Limited support for inductive theorem proving was provided by the Maude ITP tool [9], presented in 2006, although its features are not comparable to those of widespread proof assistants, its reasoning capabilities are limited to the equational part of rewriting logic, induction on the relations is not supported, and it is currently incompatible with the latest Maude releases.

* Corresponding author.
E-mail addresses: rubenrub@ucm.es (R. Rubio), ariesco@ucm.es (A. Riesco).

https://doi.org/10.1016/j.jlamp.2024.101005
Received 30 September 2023; Received in revised form 26 August 2024; Accepted 27 August 2024

Available online 30 August 2024

Lean [32] is a proof assistant based on the proposition-as-types paradigm of the calculus of inductive constructions, borrowed from Coq [3]. Notable features of Lean are powerful resolution with Haskell-like type classes, a metaprogramming framework, a multithreaded type checker, and an intuitive interactive interface. It has support for proof automation, but it is not currently as developed as that of Coq or Isabelle/HOL [38]. Moreover, the tool counts with a significant community-driven library of reusable definitions and proofs called *mathlib* [11], which has already been used to formalize non-trivial mathematical theories from undergraduate to novel research topics. The presentation in the current paper focuses on Lean 3, which was the stable version at the time of writing, but our tool also supports the new Lean version 4 [31].

This paper presents a translation from rewriting logic to the calculus of constructions with inductive types used by the Lean language. The translation is implemented in a tool that automatically derives a Lean specification from a Maude module. It can be used to reason inductively and prove theorems about Maude-specified models, their terms, and their equational and rewriting relations. The transformed specification is constructive, deterministic, and configurable and some lemmas are provided to facilitate building proofs on it. Indeed, the Lean simplifier can handle sorts and reduction to canonical form in most cases. Moreover, some optional variations can be applied to the translation in order to reduce some boilerplate arguments and lighten the verification effort.

Even though our main goal is to translate any rewriting logic theory specified in Maude to Lean, Maude specifications also involve some operational aspects that should be taken into account for the translation to be useful in practice. In addition to what can be found in the formal definition of a rewrite theory, our translation supports the `ctor` attribute of operators, some polymorphic builtins like `if_then_else_fi` and equality, and the `Bool` sort as a Lean native type. However, we do not support metatheoretic aspects like the `owise` attribute of equations, the predefined inequality operator `=/=`, the descent functions of the metalevel, and so on. Special (i.e. non-equationally-defined) operators are not supported, with some exceptions, although the user may provide its own equations for them. External objects and metainterpreters are not supported either, as they are dynamic objects without referential transparency.

This paper is based on a conference paper presented at ICFEM 2022 [43]. However, we have extended it with

1. several optional pragmatic extensions of the translation to facilitate proofs, like the conversion of derived Maude operators to Lean definitions, and the usage of the native Lean's `bool` for the Maude's `Bool` type (see Section 5);
2. extended coverage of other features of Maude specifications, like frozeness annotations [6], and some special operators;
3. the module to be translated can now be passed as a metamodule, i.e. as a term on the universal theory of `META-MODULE`;
4. support for the just-released Lean 4 in addition to Lean 3 (all features of the translation are available in Lean 4 except some helper definitions explained at the end of Section 5.1);
5. new application examples and a discussion of the effectiveness of the translation; and
6. an enhancement with additional details of the content already introduced in the conference paper.

The structure of the paper is as follows: Section 2 briefly describes the formalisms and tools involved in this work, Section 3 presents the proposed translation, and Section 4 outlines the associated tool with an example. Some pragmatical extensions and optimizations to the translation are presented in Section 5. More interesting examples are summarized in Section 6, Section 7 discusses on related work, and finally Section 8 concludes with ideas for future extension. The tool is available at github.com/fadoss/maude2lean as free software.

2. Preliminaries

In this section we describe Maude and Lean, focusing on their underlying logics.

2.1. Algebraic specification

In algebraic specification, a *many-kinded signature* $\Sigma = (K, \Sigma)$ is defined as a set of kinds K and an $(K^* \times K)$-indexed family $\Sigma = \{\Sigma_{k_1 \cdots k_n, k} : k_1 \cdots k_n \in K^*, k \in K\}$ of symbols. We say that a symbol $f \in \Sigma_{k_1 \cdots k_n, k}$ has range kind k and arity n, and receives that many arguments of kinds $k_1, \ldots, k_n$. Given a K-indexed family of variables $X = \{X_k\}$, the sets $T_{\Sigma,k}(X)$ of terms of kind k in Σ are inductively defined as the variables $x \in X_k$ and the syntactic elements $f(t_1, \ldots, t_n)$ for any $f \in \Sigma_{k_1 \cdots k_n, k}$ and any $t_i \in T_{\Sigma,k_i}(X)$ for $1 \leq i \leq n$. Terms without variables $T_{\Sigma,k}(\emptyset)$ are called *ground terms*. A *substitution* is a function $\sigma : X \to T_\Sigma(X)$ that maps each variable to a term. It is inductively extended to be applied on terms by $\overline{\sigma}(x) = \sigma(x)$ and $\overline{\sigma}(f(t_1, \ldots, t_n)) = f(\overline{\sigma}(t_1), \ldots, \overline{\sigma}(t_n))$. We write σ instead of $\overline{\sigma}$ when there is no confusion.

2.2. Rewriting logic

Maude [8] is a high-performance logical framework and specification language based on rewriting logic [26]. While rewriting logic can be built on top of any equational logic, membership equational logic [5] is the underlying formalism of choice in Maude. This logic, in addition to *sorts* and *equations*, allows users to state *membership axioms* characterizing the elements of a sort. Rewriting logic extends membership equational logic with *rewrite rules*, which stand for transitions in concurrent systems.

Membership equational logic is built on top of a many kinded signature $\Sigma = (K, \Sigma)$, but, in addition to kinds, we consider a K-indexed family $S = \{S_k\}$ of *sorts*, to which terms will be ascribed $t : s$ in the logic. A *signature* in membership equational logic (K, Σ, S) is then a many-kinded signature (K, Σ) with a K-indexed family S of sorts.

Our definition of membership equational logic follows that of [5] and introduces sorts at the very end. However, signatures in Maude are often presented as *order-sorted* ones, where a partially ordered set of sorts $(S, \leq_S)$ is introduced first, symbol signatures are defined in terms of sorts, and kinds are defined as the connected components of the $\leq_S$ relation. Indeed, kinds are not directly specified in Maude modules, but only sorts and their subsort relation. Anyhow, $\leq_S$ and the sort-based signature of symbols can be expressed as part of the theory in membership equational logic, so that both formalizations are interchangeable. Moreover, the many-kinded path is more convenient for our formalization in Lean, as we will see.

Theories in membership equational logic are defined with two classes of atomic sentences, *equations* $t = t'$ and sort *membership axioms* $t : s$, which respectively identify terms and assign them a sort. They can be combined to yield conditional Horn clauses of the form

$$t = t' \quad \text{if} \bigwedge_i u_i = u_i' \wedge \bigwedge_j v_j : s_j \qquad\qquad t : s \quad \text{if} \bigwedge_i u_i = u_i' \wedge \bigwedge_j v_j : s_j$$

Membership equational theories (Σ, E) are the combination of a signature and a set of such axioms. Equality modulo equations $(=_E) \in T_\Sigma(X) \times T_\Sigma(X)$ is defined as the finest congruence relation satisfying the Horn clauses in E, and we write $t \in T_{\Sigma,s}(X)$ if $t : s$ can be proven in E. Moreover, the set of equations is usually partitioned into *structural axioms* and normal equations, setting aside into the first subset common identities like associativity, commutativity, and identity that cannot be blindly applied without losing executability. Maude deals with them using specific algorithms. We write $=_A$ for the equational relation on these structural axioms only.

Similarly, rewriting logic theories extend the equational ones with rewrite rules of the form:

$$l \Rightarrow r \quad \text{if} \bigwedge_i u_i = u_i' \wedge \bigwedge_j v_j : s_j \wedge \bigwedge_k w_k \Rightarrow w_k'$$

Notice that rules can also be conditioned by rewriting clauses, which hold whenever the term w_k can be rewritten to a term that matches the pattern w_k'. We write $t \to_R^1 t'$ if t can be rewritten to t' by applying a single rewrite rule in one of its subterms, and define $\to_R^*$ as the reflexive and transitive closure of $\to_R^1$. Often, rewrite theories are defined together with a frozeness function ϕ that assigns to each operator the set of natural numbers corresponding to its *frozen* arguments, i.e., those in which rewriting with rules is forbidden.

The rule rewriting relation is defined by means of the following axioms [26]:

1. *Reflexivity*. For each $t \in T_\Sigma(X)$, $\vdash t \to t$.
2. *Equality*. $u \to v, u =_E u', v =_E v' \vdash u' \to v'$.
3. *Congruence*. For each $f : k_1 \cdots k_n \to k$ in Σ, and $t_1, t_1', \ldots, t_n, t_n' \in T_\Sigma(X)$,

$$\frac{t_1 \to t_1' \quad \cdots \quad t_n \to t_n'}{f(t_1, \ldots, t_n) \to f(t_1', \ldots, t_n')}$$

   where $t_k = t_k'$ if $k \in \phi(f)$.
4. *Replacement*. For each rule $r : t \to t'$ if $\bigwedge_i u_i = u_i' \wedge \bigwedge_j v_j : s_j \wedge \bigwedge_k w_k \to w_k'$ with variables $\{x_1, \ldots, x_n\}$ and each substitution such that $\sigma(x_i) = p_i$:

$$\frac{\begin{array}{cccccc} p_1 \to p_1' & \cdots & p_n \to p_n' & \cdots & \sigma(u_i) =_E \sigma(u_i') & \cdots \\ \cdots & \sigma(w_k) \to \sigma(w_k') & \cdots & \cdots & \sigma(v_j) \in T_{\Sigma,s_j} & \cdots \end{array}}{\sigma(t) \to \sigma[x_1 \,/\, p_1', \ldots, x_n \,/\, p_n'](t')}$$

   where $p_k = p_k'$ if x_k is in a frozen position in t or t'.
5. *Transitivity*. $t_1 \to t_2, t_2 \to t_3 \vdash t_1 \to t_3$.

One-step rewrites are those derivable from rules (1-4) with at least one application of rule (4), but allowing all rules (1-5) in the derivation of sequents. Sequential rewrites are those that can be derived from (1-5) where the premises $p_i \to p_i'$ of (4) are not derived with (3) and at most one premise of (4) is derived with (3). Both relations, $\to^1$ and $\to^*$, are sequential and $\to^1$ is also one-step.

2.3. Maude

In Maude, membership equational logic specifications are defined in functional modules, with syntax `fmod NAME is SENS endfm`, where `NAME` is the module name and `SENS` is a set of sentences. Let us illustrate the Maude syntax with a specification of the natural numbers[1] with addition and distinguishing even numbers. First, the `PEANO` module is opened, declaring the sorts `Nat` (for natural numbers), `Even` (for even numbers), and `NzNat` (for non-zero natural numbers) by using the `sorts` keyword:

[1] Maude provides a predefined module `NAT` for natural numbers, although their operations are not defined by means of Maude equations but at the C++ level.

```
fmod PEANO is
  sorts Nat Even NzNat .
```

Because both even and non-zero numbers are particular cases of natural numbers, we can use a `subsort` declaration to state it:

```
  subsorts NzNat Even < Nat .
```

We define how terms of these sorts are built by specifying operators:

```
  op 0 : -> Even [ctor] .
  op s_ : Nat -> NzNat [ctor] .
```

An operator is declared with the keyword `op` followed by its name (where underscores stand for placeholders), a colon, the arity (which is empty in the case of constants), an arrow `->`, its coarity, and a set of attributes enclosed in square brackets. In this case, we have used only the `ctor` attribute, telling that the operators are *constructors* of the corresponding sorts. Finally, observe that the constant `0` has sort `Even`, while the successor (`s_`) receives any natural number and returns a non-zero natural number. Similarly, the addition operator would be defined as:

```
  op _+_ : Nat Nat -> Nat .
```

Typically, we would have added `[assoc comm id: 0]` to indicate that the sum is associative (`assoc`), commutative (`comm`), and has `0` as identity element (`id: 0`). However, we will later prove that these properties of the operator follow from its equational definition. Operators can also take other attributes like `frozen(`$i_1 \cdots i_m$`)` to indicate that no rewrite is allowed in the specified arguments, i.e. that $\phi(f) = \{i_1, \ldots, i_m\}$. Next, we define some variables of sort `Nat` that will be used later in membership axioms and equations:

```
  vars N M : Nat .
```

Membership axioms can be declared with syntax `mb` (`cmb` for *conditional* membership axioms) to specify the sort of terms with more flexibility than in operator signatures. In our case, even numbers are recursively identified as:

```
  cmb s s N : Even if N : Even .
```

It is important to note that subsorts (which can be understood as primitive membership axioms, `X:`s $:$ t if $s < t$) determine the kinds of the signature, which are identified with the connected components of each sort S under the subsort relation. The kind of the sort s is written `[`s`]`, and Maude introduces an additional sort with the same name as the kind for its error terms, that is, those that cannot be assigned a proper sort. In our case, the sorts `Nat`, `NzNat`, and `Even` have the same kind because they are related by a subsort. We identify this kind by `[Nat]` because `Nat` is the top sort in the relation.

From the Maude point of view, and this will be relevant in our translation, operator declarations are internally transformed to both a kind-level declaration of the symbol and a membership axiom assigning the result sort to any application of the operator whose arguments have the correct sorts. For example, the addition operator above would be transformed into:

```
  op _+_ : [Nat] [Nat] -> [Nat] .
  mb N:Nat + M:Nat : Nat .
```

where the sorts of the variables follow those of the original declaration.

Equations are introduced with syntax `eq` (`ceq` for conditional equations). In the example, the equations for addition can be defined as follows by discriminating constructors in the second argument:

```
  eq N + 0 = N .
  eq N + s M = s (N + M) .
endfm
```

Functional modules are closed with the `endfm` keyword.

Rewrite theories are specified in *system modules*, with syntax `mod NAME is SENS endm`, where `NAME` is the module name and `SENS` is a set of sentences, including those from functional modules plus rewrite rules. The module `PEANO-WITH-RULE` below imports the `PEANO` module[2] and introduces a rule, with label `cancel`, for transforming an odd number into the previous one:

```
mod PEANO-WITH-RULE is
  protecting PEANO .
  var N : Nat .
  crl [cancel] : s N => N if N : Even .
endm
```

Rules are inserted with the `rl` and `crl` keywords, and `=>` is used instead of `=` to separate both sides.

The Maude interpreter supports several commands for executing Maude specifications. For example, terms can be reduced to their normal forms modulo equations with `reduce`.

[2] Module can be imported in Maude in four different modes: `protecting` (no junk and no confusion), `extending` (no confusion), `generated-by` (no junk), and `including` (no guarantees). Maude makes no attempt to check these promises. See [7, §7.1] for a thoughtful explanation of these importation modes.

```
Maude> reduce s s 0 + s s s 0 .
rewrites: 4
result NzNat: s s s s s 0
```

Rules are executed exhaustively with the `rewrite` command.

```
Maude> rewrite s s 0 + s s s 0 .
rewrites: 9
result Even: 0
```

And for searching in the rewrite graph, there is the `search` command.

```
Maude> search s s 0 + s s s 0 =>* s 0 .

Solution 1 (state 4)
states: 5  rewrites: 8
empty substitution

No more solutions.
```

This tells us that `s 0` (that is, 1) can be reached by rewriting from the initial term $2 + 3$. In general, the right-hand side can be a pattern with variables, conditions can be added, and the intermediate steps can be inspected with `show path` and `show graph` [7].

2.4. Dependent type theory and Lean

Interactive theorem provers are tools that assist humans in writing formal specifications and proving theorems about them. Usually, they do not only check the proof steps specified by the human, but also provide automation features to simplify routine work. Well-known examples are Isabelle/HOL [34], Agda [35], Coq [3], and Lean [32]. The logical foundations of the last three are extensions of Martin-Löf dependent type theory with inductive types, where propositions are seen as types whose elements are their proofs.

Given a universe of types $\mathcal{U}$, for any type A and for any A-indexed collection $B : A \to \mathcal{U}$ of them, two dependent type constructors $\Pi_{x:A} B(x)$ and $\Sigma_{x:A} B(x)$ are considered. Intuitively, elements of $\Pi_{x:A} B(x)$ are functions that given an $a : A$ produce a result $b : B(a)$, where the type of the result may depend on the argument itself and not only its type. If there is no such a dependency, we obtain a standard functional type $A \to B$. On the other hand, elements of $\Sigma_{x:A} B(x)$ are seen as pairs (a, b) for some $a : A$ and $b : B(a)$, whose independent case is the Cartesian product $A \times B$. Inductive dependent types $B(a_1, \ldots, a_n)$ are given by a finite collection of constructors $C_k : \Pi_{a_1:A_1} \cdots \Pi_{a_n:A_n(a_1,\ldots,a_{n-1})} B(a_1, \ldots, a_n)$ with arguments from other types and from the type itself. Constructors with independent types can be seen as the operators of a many-kinded signature (as defined in Section 2.1) and terms are built using them.

From a logical point of view, as expressed in the Curry-Howard correspondence, types are seen as propositions, an element of type A is a proof that A holds, a function/implication $A \to B$ yields a proof of B given a proof of A, and so on. Elements of $\Pi_{x:A} B(x)$ give a proof of $B(a)$ for every a, so the type can be seen as a universally quantified formula and we also write $\forall x : A, B(x)$. Elements of $\Sigma_{x:A} B(x)$ give an element a and a proof of $B(a)$, so this type can be understood as an existentially quantified formula and written $\exists x : A, B(x)$. Equality can be defined as an inductive dependent type `eq A B` with a single constructor `refl` of type `eq A A`, so that equality only holds for identical terms. Other relations can be defined similarly.

Lean [32] is a modern interactive theorem prover founded on the *calculus of inductive constructions*, a dependent-type formalism borrowed from Coq [3]. Developed by Microsoft Research, it counts with a unified and extensive library of mathematics *mathlib* [11] maintained by its community, used to formalize quite complex mathematical theories. The term language of Lean is a dependent λ-calculus with variables, constants, function application, λ-abstraction, function spaces, metavariables, and macros. There is a special type `Prop` for propositions, and a hierarchy of types `Type u` parameterized by universes.[3] Inductive types, families, and definitions can be declared with the `inductive` and `def` keywords, axioms can be established with `axiom`, and results can be provided with `lemma` and `theorem`. In order to prove them, a collection of tactics are available, like `exact` to provide an element for the target type, `rw` to rewrite with an identity, `simp` to apply the Lean simplifier, `apply` to apply a hypothesis, `cases` to consider each case of an inductive type or obtain the witness of an existential claim, and `induction` to proceed by induction. In Sections 4 and 6.1, we show step-by-step examples of proofs using these tactics in Lean.

Following the previous example, Peano numbers can be defined in Lean with this inductive type:

```
inductive nat : Type
  | zero : nat
  | succ : nat → nat
```

The sum of two natural numbers can then be defined with `def` as

```
def sum : nat → nat → nat
  | n zero     := n
  | n (succ m) := succ (n + m)
```

[3] Type universes form a countable non-cumulative hierarchy $\mathcal{U}_0, \mathcal{U}_1, \mathcal{U}_2, \ldots$ to avoid Russell-like paradoxes.

or equivalently as

```
def sum (n : nat) : nat → nat
  | zero     := n
  | (succ m) := succ (n + m)
```

where we have put the first operand as a named argument of the definition. Mutually recursive definitions are introduced with the `mutual def` or `mutual inductive` keywords, like

```
mutual def even, odd
with even : nat → Prop
  | zero := true
  | (succ n) := odd n
with odd : nat → Prop
  | zero := false
  | (succ n) := even n
```

Notice the use of the basic type `Prop` for the result of `even` and `odd`. The induction principle can be formulated with a formula of that type as an axiom

```
axiom induction_principle (p : nat → Prop) (base_case : p zero)
  : (∀ n, p n → p (succ n)) → ∀ n, p n
```

and the fact that $0 + n = n$ can be stated and proven using a lemma:

```
lemma zero_sum (n : nat) : sum zero n = n :=
begin
  induction n, --- etc.
end
```

More complex parametric inductive types called *inductive families* can be defined, like generic vectors of a given size[4]

```
inductive vector (α : Type u) : nat → Type u
  | nil                                  : vector zero
  | cons {n : nat} (a : α) (v : vector n) : vector (succ n)
```

or equality as explained before

```
inductive eq {α : Sort u} (a : α) : α → Prop
  | refl : eq a -- eq a a (the first a is above)
```

where for a given `a` only `eq a a` is populated by `refl` [1]. Mutually inductive families will be used in the rest of the paper to translate the various relations involved in rewriting logic specifications. The same constructs are available in Lean 4, although with a slightly different syntax.

Finally, we should mention the Lean simplifier, since this is a basic tool to write proofs in an affordable manner. It can simplify terms or automatically advance proofs by using the marked axioms and lemmas of the theory. Lean uses attributes to indicate which definitions and lemmas should be used by the simplifier. They are specified with the decorator `@[attr]` before the definition, or with `attribute [attr]` followed by a list of names of definitions or lemmas, as a separate instruction. The main ones are `simp` for using the axiom or lemma as a simplification rule, `symm` for introducing a more delicate symmetry or commutativity axiom, and `congr` for marking congruence lemmas and allow relations to be applied inside arguments.

3. The translation

Given a rewrite theory $\mathcal{R} = (\Sigma, A \cup E, \phi, R)$ where $\Sigma = (K, \Sigma, S)$ is a K-kinded signature with a K-indexed family S of sorts, A is a set of equations specifying builtin structural axioms like commutativity, E is a set of normal equations and membership axioms, ϕ is the frozeness function, and R is a set of rewrite rules, everything given by a Maude module M, we define the following specification in Lean. This general procedure is illustrated with a particular example in Section 4.

In a nutshell, terms in the Maude signature are represented as elements of some inductive types, one for each kind $k \in K$, and the sort membership, equational, and rewriting relations are inductively defined on them. More precisely, we declare an inductive type `MSort` of constants enumerating the sorts S in the module, and for each kind $k \in K$

1. an inductive type k with a constructor for every operator $f \in \bigcup_{d \in K^*} \Sigma_{d,k}$ to represent the terms $T_{\Sigma,k}(X)$ of kind k. By default, the inductive type is named `k`s where s is the most general sort of the kind k (if there are many, the first one to be declared).

[4] Notice that curly brackets are used in the argument `n` of `cons`. This means that we should write `cons zero nil` without passing n, because $n = 0$ can be inferred from the type of `nil`.

2. a binary relation k`.has_sort` of type $k \to$ `MSort` $\to$ `Prop` where k`.has_sort` t s means $t \in T_{\Sigma,s}(X)$. In Lean and in this paper, we use the overloaded infix notation $t \rhd s$.
3. two binary relations k`.eqa` and k`.eqe` of type $k \to k \to$ `Prop` for the equivalence relation of terms modulo structural axioms and modulo equations, respectively. Again, we use the overloaded infix notation `=A` and `=E` for them.
4. k`.rw_one` and k`.rw_star` of type $k \to k \to$ `Prop` for the one-step sequential rule-rewrite relation and for its reflective and transitive closure. In infix form, we write `=>1` and `=>*`.

We detail the concrete definition of these elements in the rest of the section. Because these relations always have cross dependencies between them and with their counterparts for other kinds, their definitions should be mutually inductive.

Regarding the representation of terms, symbols $f \in \Sigma_{d_1 \cdots d_n,k}$ in $\mathcal{R}$ are mapped one-to-one to their homonym constructors $f : d_1 \to \cdots \to d_n \to k$ of the inductive type k, so there is an isomorphism of Σ-algebras between $T_{\Sigma,k}(\emptyset)$ and the elements of the Lean type k. The translation does not distinguish by default between constructor and defined operators, since marking operators with the `ctor` attribute is not common practice, and deciding whether they are completely defined is non-trivial [22]. Unlike other works [23,29] that represent each Maude sort as a type with coercions to pass from the subsort to the supersort, our translation works with types at the kind level and handles sort membership with the explicit `has_sort` relation, as explained in the next paragraphs.

Statements in Maude (explicit membership axioms, equations, and rewrite rules) are translated under the same general principles. Every statement takes the form "φ if $\varphi_1 \wedge \cdots \wedge \varphi_n$" where $\varphi, \varphi_1, \ldots, \varphi_n$ are atomic sentences that may contain some variables $a_1, \ldots, a_m$ of sorts $s_1, \ldots, s_m$. The sort-membership sentence $l : s$ in Maude is translated to $l \rhd s$ in Lean, the equality sentences $l = r$ and $l := r$ to l `=E` r,[5] and the rewriting sentence l `=>` r to l `=>*` r in the condition and to l `=>1` r in the main sentence φ. Thus, we translate the above statement to the chain of implications or the dependent functional type

$$\begin{aligned} \forall\, a_1 : k_1, \ldots, a_m : k_m, \quad & a_1 \rhd s_1 \to \cdots \to a_m \rhd s_m \to \\ & T(\varphi_1) \to \cdots \to T(\varphi_n) \to T'(\varphi) \end{aligned} \tag{$\star$}$$

where $T(\psi)$ is the translation of the atomic sentence ψ (T' coincides with T except that l `=>` r is translated to l `=>1` r, as explained above). This chain is logically equivalent to $(\bigwedge_{i=1}^{m} a_i \rhd s_i \wedge \bigwedge_{j=1}^{n} T(\varphi_j)) \to T(\varphi)$. In addition to its own condition, statements also include sort-membership obligations for the variables, since their type in Lean is only associated to a kind.

The schema in the previous paragraphs is used to specify the sort membership, equational, and rewriting relations enumerated in (2-4). Each relation is declared as an *inductive family* in Lean, in other words, as a dependent parameterized type defined by an inductive enumeration of cases. The only self-contained relation is equivalence modulo structural axioms `=A`, which is given by the following constructors

- `refl` (reflexivity) of type $\forall\, a : k,\ a$ `=A` a.
- `symm` (symmetry) of type $\forall\, a, b : k,\ a$ `=A` $b \to b$ `=A` a.
- `trans` (transitivity) of type $\forall\, a, b, c : k,\ a$ `=A` $b \to b$ `=A` $c \to a$ `=A` c.
- f`_assoc` of type $\forall\, a, b, c : k,\ f\ a\ (f\ b\ c)$ `=A` $f\ (f\ a\ b)\ c$ for every associative operator f in k.
- f`_comm` of type $\forall\, a, b : k,\ f\ a\ b$ `=A` $f\ b\ a$ for every commutative operator f.
- f`_left_id` of type $\forall\, a : k,\ f\ e\ a$ `=A` a for every operator f with left identity element e. And a similar constructor for f`_right_id`.
- f`_idem` of type $\forall\, a : k,\ f\ (f\ a) = f\ a$ for every idempotent operator f.
- `eqa_`f (congruence) of type $\forall_{i=1}^{n} a_i : k_i,\ a_1$ `=A` $b_1 \to \cdots \to a_n$ `=A` $b_n \to f\ a_1 \cdots a_n$ `=A` $f\ b_1 \cdots b_n$ for every operator f of arity n in the kind. Notice that `=A` may be the counterpart relation k_i`.eqa` of other kind.

Although we have written the types of the constructors as first order formulas for readability, they represent dependent types where universal quantification $\forall a : k$ can be replaced by $\Pi_{a:k}$. For example, `refl` has type $\Pi_{a:k}\ k$`.eqa` $a\ a$. Intuitively, the constructors of the inductive relation are the closed set of axioms defining it. A statement l `=A` r is true iff it can be derived from these axioms, and we can reason by induction whether a statement holds or not. Technically, terms of type l `=A` r built with these constructors are proofs that the relation holds for l and r, and inhabited types are exactly the true propositions. For example, if `f` is an associative and commutative symbol, a proof of `f` a (`f` $b\ c$) `=A` `f` (`f` $b\ a$) c is

```
trans f_assoc (eqa_f f_comm refl)
```

where we have omitted the universally quantified variables in the constructor applications (indeed, they are defined as implicit arguments in Lean, which are inferred by the tool from the context) and their namespaces for readability. The reader can check it by writing and operating the types of the elements of the term above as they appear in the previous enumeration.

[5] The atomic sentence $l := r$ is equivalent to $l = r$ but allowing free variables in the left-hand side to be instantiated by matching, which does not make a difference here.

The sort-membership relation k . has_sort, written $\rhd$ in infix form, consists of the following constructors

- subsort of type $\forall\, t : k,\ r, s : \texttt{MSort},\ \texttt{subsort}\ r\ s \to t \rhd r \to t \rhd s$ where $\texttt{subsort} : \texttt{MSort} \to \texttt{MSort} \to \texttt{Prop}$ is the generator of the subsorting relation specified with subsort declarations in Maude.
- f _decl of type $\forall_{i=1}^{n}\, t_i : k,\ t_1 \rhd s_1 \to \cdots \to t_n \rhd s_n \to (f\ t_1 \cdots t_n) \rhd s$ for each operator declaration $f : s_1 \cdots s_n \to s$ as its implicit membership axiom.
- For each mb or cmb statement in Maude, a constructor whose type is derived as in (⋆).

Equivalence modulo equations and structural axioms =E, also written k . eqe, is given by the constructors

- from_eqa of type $\forall\, a, b : k,\ a =\texttt{A}\ b \to a =\texttt{E}\ b$ stating that =A is finer than =E.
- Both symm and trans as defined for =A. However, refl follows from the constructors of =A and from_eqa, so it is not introduced as a constructor.
- For each symbol f in the kind, a congruence constructor eqe_f with the same definition as eqa_f where =A has been replaced by =E.
- For each eq and ceq statement in Maude, a constructor whose type is derived as in (⋆).

Standard axioms like reflexivity and transitivity are marked with builtin attributes in the Lean specification to facilitate their use within the system and by its automation infrastructure.

Rewriting relations are specified in a very similar way. We first define the one-step rule relation k . rw_one (or =>1 in infix form) and then make k . rw_star (=>*) its reflective and transitive closure. However, since conditional rules with reachability premises are allowed, the definitions of =>1 and =>* should be simultaneous and mutually recursive. The one-step relation is given by the following constructors

- eqe_left to apply a rewrite modulo equations on the left, of type $\forall\, a, b, c : k,\ a =\texttt{E}\ b \to b =\texttt{>1}\ c \to a =\texttt{>1}\ c$.
- eqe_right to apply a rewrite modulo equations on the right, of type $\forall\, a, b, c : k,\ a =\texttt{>1}\ b \to b =\texttt{E}\ c \to a =\texttt{>1}\ c$.
- For each operator f of arity n and each $0 \le i < n$, a constructor sub_f_i to apply a rule inside the i^{th} argument of f, whose type is $\forall_{j=0}^{n}\, t_j, t_i' : k,\ t_i =\texttt{>1}\ t_i' \to (f\ t_1 \cdots t_i \cdots t_n) =\texttt{>1}\ (f\ t_1 \cdots t_i' \cdots t_n)$. However, we omit the definition of sub_f_i if the i^{th} of f is frozen, $i \in \phi(f)$.
- For each rl and crl statement in Maude, a constructor whose type is derived as in (⋆).

On the other hand, =>* is defined with three constructors

- refl (reflexivity) with type $\forall\, a, b : k,\ a =\texttt{E}\ b \to a =\texttt{>*}\ b$,
- step of type $\forall\, a, b : k,\ a =\texttt{>1}\ b \to a =\texttt{>*}\ b$, and
- trans for transitivity, whose type coincides *mutatis mutandis* with the equivalent constructors for previous relations.

Some basic and useful properties when elaborating proofs are a consequence of the previous axioms, like the extension of the sub_f_i axioms for the =>* relation. In order to facilitate proofs, these and other properties are generated and proven as lemmas by our translation, as explained in Section 5.1. The following proposition establishes the soundness of the translated specification for proving theorems about rewriting logic specifications in Lean.

Proposition 1. *Given a rewrite theory $\mathcal{R}$, two terms $t, t' \in T_{\Sigma,k}(X)$ for some kind k, and a sort $s \in S_k$, and with $\vdash$ A meaning that the type A is inhabited:*

1. *There is an one-to-one mapping T from $T_\Sigma(X)$ to the Lean inductive terms of type* kk *for $k \in K$.*
2. $t \in T_{\Sigma,s}(X)$ *iff* $\vdash T(t) \rhd T(s)$.
3. $t =_A t'$ *iff* $\vdash T(t) =\texttt{A}\ T(t')$
4. $t =_E t'$ *iff* $\vdash T(t) =\texttt{E}\ T(t')$
5. $t \to_R^1 t'$ *iff* $\vdash T(t) =\texttt{>1}\ T(t')$
6. $t \to_R^* t'$ *iff* $\vdash T(t) =\texttt{>*}\ T(t')$

Proof. (1) For each kind k of the Maude signature, an inductive type kk is defined in Lean with a constructor $f : \texttt{k}k_1 \to \cdots \to \texttt{k}k_n \to \texttt{k}k$ for each operator $f : k_1 \cdots k_n \to k$ in Maude. Terms are inductively defined in the same way for both formalisms, so $T(k) = \texttt{k}k$ and $T(f(x_1, \cdots, x_n)) = f\ T(x_1) \cdots T(x_n)$ is a one-to-one mapping.

(2-4) These claims follow from the fact that proof terms of membership equational logic [5] for $t =_A t'$, $t =_E t'$, and $t : s$ are in a one-to-one correspondence with the elements of $T(t) =\texttt{A}\ T(t')$, $T(t) =\texttt{E}\ T(t')$, and $T(t) \rhd s$. Indeed, the inductive constructors of these types and the deduction rules of membership equational logic are essentially the same (see Figure 4 in [5]). We omit the inductive proof for brevity and since it is much similar to the one below for the rewriting relations.

(5-6) The rule rewriting relations $\to^1$ and $\to^*$ are defined in [27, §3.1] (and in Section 2.2) by the finite application of five deduction rules with some restrictions. We will prove by structural induction that, for any two terms $t, t' \in T_{\Sigma,k}(X)$, there is a one-to-one correspondence between the proof trees of $t \to^1 t'$ (respectively, $t \to^* t'$) and the terms of type k . rw_one $T(t)$ $T(t')$ (respectively, k . rw_star $T(t)$ $T(t')$). First, given a proof tree, we show the corresponding proof term.

1. The tree $\vdash t \to t$ produced by rule (1) can only prove the statement $t \to^* t$, since there is no application of rule (3). An element of type $T(t)$ `=>*` $T(t)$ is `k.rw_star.refl (k.eqe.from_eqa k.eqa.refl)`.
2. For the *equality* rule, we have a term α of sort $T(u)$ `=>1` $T(v)$ (the same with `=>*`) by induction hypothesis. Since $u =_E u'$ and $v =_E v'$, we have terms β_u of type $T(u)$ `=E` $T(u')$ and β_v of type $T(v)$ `=E` $T(v')$. An element of type $T(u')$ `=>1` $T(v')$ is `k.rw_one.eqe_left` β_u (`k.rw_one.eqe_right` α β_v). An element of type $T(u')$ `=>*` $T(v')$ is

$$\begin{aligned} &\texttt{k.rw\_star.trans} \quad (\texttt{k.rw\_star.refl}\ \beta_u) \\ &\qquad\qquad\qquad\quad (\texttt{k.rw\_star.trans}\ \alpha\ (\texttt{k.rw\_star.refl}\ \beta_v)). \end{aligned}$$

3. For the *congruence* rule, suppose $f(t_1,\dots,t_n) \to^1 f(t'_1,\dots,t'_n)$, so exactly one of the derivations $t_i \to t'_i$ uses (3), say the j^{th} one. This implies $t_i =_E t'_i$ for $i \neq j$ and $t_j \to^1 t'_j$. By induction hypothesis, there is a term α_j of type $T(t_j)$ `=>1` $T(t'_j)$ and terms α_i of type $T(t_i)$ `=E` $T(t'_i)$. The element

$$\begin{aligned} &\texttt{k.rw\_one.eqe\_right}\ (\texttt{k.rw\_one.sub\_}f_j\ \alpha_j) \\ &(\texttt{k.eqe.eqe\_}f\ \ \alpha_1 \cdots (\texttt{k}_j\texttt{.eqe.from\_eqa}\ \texttt{k}_j\texttt{.eqa.refl}) \cdots \alpha_n) \end{aligned}$$

   has type $f(T(t_1),\dots,T(t_n))$ `=>1` $f(T(t'_1),\dots,T(t'_n))$. The $\to^*$ case can be proven similarly using the fact that

$$t_j \texttt{ =>* } t'_j \to f(\dots,t_j,\dots) \texttt{ =>* } f(\dots,t'_j,\dots),$$

   which is proven as a lemma `sub_star` in the `infer_sub_star.lean` file of the repository [44]. Notice that k`.rw_one.sub_`f_i is always defined, because $i \notin \phi(f)$ for the congruence rule to be applied with a rewrite on the i-th argument.
4. For the *replacement* rule, given a rewrite rule $r : t \to t'$ (where the sort membership of variables can be assumed to be given by explicit conditions) and a substitution σ, if $\sigma(t) \to^1 \sigma[x_i/p'_i](t')$, by induction hypothesis, there are elements α_i of type $T(\sigma(u_i))$ `=E` $T(\sigma(u'_i))$, β_j of type $T(\sigma(v_j))$ $\rhd$ s_j, γ_k of type $T(\sigma(w_k))$ `=>*` $T(\sigma(w'_k))$, and δ_l of type $T(p_l)$ `=E` $T(p'_l)$, since (3) cannot be applied in $p_l \to p'_l$. The constructor `rl_`r, whose type is derived from r by ($\star$), yields a term ξ of type $T(\sigma(t))$ `=>1` $T(\sigma(t'))$ when applied the premises α_i, β_j, and γ_k in the appropriate order. Finally, we apply `eqe_right` to ξ and an appropriate term of type $T(\sigma(t'))$ `=E` $T(\sigma[x_i/p'_i](t'))$ built with `eqe_`f constructors and δ_l terms. The $\to^*$ case follows by applying `rw_star.step` to ξ, and then the lemma `sub_star` with the δ_l proof terms of type p_l `=>*` p'_l.
5. For the *transitivity* rule, which can only be applied to derive $t_1 \to^* t_3$, there are proof terms α of type $T(t_1)$ `=>*` $T(t_2)$ and β of type $T(t_2)$ `=>*` $T(t_3)$ by induction hypothesis. k`.rw_star.trans` α β has type $T(t_1)$ `=>*` $T(t_3)$.

Conversely, proof trees in rewriting logic can be built for any term of type $T(t)$ `=>1` $T(t')$ or $T(t)$ `=>*` $T(t')$ in Lean. A term of type $T(t)$ `=>1` $T(t')$ must be one of

- `rw_one.eqe_left` α β with α of type $T(t)$ `=E` a and β of type a `=>1` $T(t')$. a is a term of type `k`k, so there is a $t_m \in T_{\Sigma,k}$ such that $a = T(t_m)$. Then, by the fourth item of this proposition, $t =_E t_m$, and by induction hypothesis, $t_m \to^1 t'$ holds and there is a proof tree for $t_m \to t'$ satisfying the restrictions of a one-step sequential rewrite. Hence, we can apply rule (2) of rewriting logic with it to conclude.
- `rw_one.eqe_right` β α is completely analogous.
- the congruences `sub_`f_i α with α of type a `=>1` b follow from (1) and (3). Again, there is t_i and t'_i such that $a = T(t_i)$ and $b = T(t'_i)$, and then by induction hypothesis $t_i \to^1 t'_i$, i.e. $t_i \to t'_i$ with a single application of (4) in its proof tree (except for rewriting conditions). Moreover, we have $t_j \to t_j$ by rule (1) of rewriting logic. Hence, we have $f(t_1,\dots,t_i,\dots,t_n) \to^1 f(t_1,\dots,t'_i,\dots,t_n)$ by applying (3) with a single application of (4) in the i-th argument. Notice there is no axiom `sub_`f_i if $i \in \phi(f)$, so no rewrites are applied in frozen arguments.
- and the concrete rule constructors follow from rule (4). Again, Lean terms can be written as $T(t)$ for some t by the surjectivity of T on the inductive term of the corresponding kind. Then, a proof tree for (4) can be built gathering the induction hypotheses, and equational and sort-membership premises by items (3) and (4) of this proposition.

For the $\to^*$ relation,

- `rw_star.refl` α with α of sort $T(t)$ `=E` $T(t')$ follows from rules (1) and (2). Indeed, we have $t =_E t'$ by item (4) of this proof, $t' =_E t'$ by the reflexivity of $=_E$, and $t \to t$ by rule (1) of rewriting logic. Hence, we have $t \to^* t'$ by rule (2) of rewriting logic.
- `step` follows directly from the fact that $\to^1$ is defined as a more restrictive version of $\to^*$ in rewriting logic,
- and `trans` α β with α of type $T(t)$ `=>*` a and β of type a `=>*` $T(t')$ follows from rule (5). Indeed, there is a t_m such that $T(t_m) = a$, and by the induction hypothesis $t \to^* t_m$ and $t_m \to^* t'$. The transitivity rule of rewriting logic let us conclude $t \to t'$ as a sequential rewrite, and so $t \to^* t$. □

4. The translation tool

The translation explained in the previous section is implemented in a Python package using the `maude` Python library [41]. Given a Maude source file and a selected module, obtaining its translation to Lean is simply invoking the command

`./maude2lean` ⟨*Maude source* | *specification*⟩ [`-o` ⟨*output file*⟩]

Instead of a Maude file, the user can also provide a JSON, YAML, or TOML specification of the translation, allowing some customization of the Lean output. For instance, custom renaming can be specified for Maude names, the notation used for relations can be chosen, the optimizations explained in Section 5 can be enabled or disabled, and so on. All these parameters are documented in the repository of the tool [44], shown with its `-help` flag, and enumerated in Appendix A.

In order to build proofs using the axioms of the translated specification, it is important to know their names. The tool uses the predictable and systematic nomenclature of Section 3, although the generated Lean code is organized and includes comments to easily locate the desired elements. Statements are named after their labels prefixed by `mb_`, `eq_`, or `rl_` to avoid ambiguities. In the absence of a label, the name of the top symbol in the left-hand side is used. In case multiple statements are assigned the same name with the just described procedure, a number is appended according to the position of the statements in the file. All symbols and axioms are declared inside the namespace of its corresponding kind.

For example, let us show the result of the translation for the system module `PEANO-WITH-RULE` in Section 2. The generated Lean code starts with the inductive type `MSort` enumerating the sorts in the module. This list includes the sort `Bool` because the predefined `BOOL` module is implicitly imported by default into any Maude module.

```
inductive MSort
  | Bool
  | Nat
  | Even
  | NzNat
```

Moreover, the generator of their subsort relation is specified as an inductively-defined relation:

```
def subsort : MSort → MSort → Prop
  | MSort.NzNat MSort.Nat := true
  | MSort.Even  MSort.Nat := true
  | _           _         := false
```

The definition of the kinds and their operators follows. In the `PEANO-WITH-RULE` module there is only two kinds, `[Nat]` and `[Bool]`, written here as `kNat` and `kBool`. Here is the definition of `kNat`.

```
inductive kNat
  | zero
  | s    : kNat → kNat
  | sum  : kNat → kNat → kNat
```

Notice that the `+` operator has been renamed as `sum` because symbols are not allowed in Lean identifiers.[6] The specification of the signature concludes with some auxiliary definitions: a function `kind` to map sorts to kinds, and a predicate `ctor_only` to recognize terms composed only of constructor symbols.

```
def kind : MSort → Type
  | MSort.Bool  := kBool
  | MSort.Nat   := kNat
  | MSort.Even  := kNat
  | MSort.NzNat := kNat
```

```
mutual def kNat.ctor_only, ...
with kNat.ctor_only : kNat → Prop
  | kNat.zero  := true
  | (kNat.s a) := a.ctor_only
  | _          := false
```

These are not needed in the rest of the translation (indeed, the latter can be disabled with the `with-ctor-predicate` option), but they can be useful for the user while writing proofs.

We then find the specification of the equational relations: *k*`.eqa` for equality modulo axioms, and the potentially mutually-related *k*`.has_sort` for sort membership, and *k*`.eqe` for equality modulo equations (and axioms).

```
mutual inductive kBool.has_sort, kNat.has_sort, kBool.eqe, kNat.eqe
```

The following is the inductive definition of the sort-membership relation for `kNat`. As explained in Section 3, every sort membership relation includes the fixed `subsort` constructor, `zero_decl`, `s_decl`, and `sum_decl` are derived from the corresponding operator declarations, and `mb_s` is the explicit membership axiom `cmb s s N : Even if N : Even .`

[6] Symbols and infix notation can be declared in Lean with the `notation` or `infix` statements [1, §6.6], or by implementing type classes that declare notation for all instances, like `has_le` for ≤.

```
with kNat.has_sort : kNat → MSort → Prop
  | subsort {t a b} : subsort a b → kNat.has_sort t a
                                  → kNat.has_sort t b
  | zero_decl : kNat.has_sort kNat.zero MSort.Even
  | s_decl {a : kNat} : kNat.has_sort a MSort.Nat →
                        kNat.has_sort (kNat.s a) MSort.NzNat
  | sum_decl {a₀ a₁ : kNat} : kNat.has_sort a₀ MSort.Nat →
                              kNat.has_sort a₁ MSort.Nat →
                              kNat.has_sort (kNat.sum a₀ a₁) MSort.Nat
  | mb_s {n} : kNat.has_sort n MSort.Even →
               kNat.has_sort (kNat.s (kNat.s n)) MSort.Even
```

The variables involved in each constructor are declared between curly braces, which make them *implicit* variables that are inferred by Lean from the other arguments, as mentioned in Section 2.4. Equality modulo equations is given by the following constructors, among others, where `eqe_s` is the congruence axiom for the successor symbol and the last two corresponds to the equations for the sum.

```
with kNat.eqe : kNat → kNat → Prop
  | from_eqa {a b} : kNat.eqa a b → kNat.eqe a b
  …
  | eqe_s {a b : kNat} : kNat.eqe a b → kNat.eqe (kNat.s a) (kNat.s b)
  …
  | eq_sum₀ {n} : kNat.has_sort n MSort.Nat →
                  kNat.eqe (kNat.sum n kNat.zero) n
  | eq_sum₁ {n m} : kNat.eqe (kNat.sum n (kNat.s m))
                             (kNat.s (kNat.sum n m))
```

Finally, the rewriting relations, *k*`.rw_one` and *k*`.rw_star` are defined in a similar way, with the fixed `eqe_left` and `eqe_right`, the constructors for rewriting inside an argument like `sub_s` and `sub_sum`$_0$, and the constructor `rl_cancel` for the only rule `crl [cancel] : s N => N if N : Even .` in the module.

```
mutual inductive kBool.rw_one, kNat.rw_one, kBool.rw_star, kNat.rw_star
…
with kNat.rw_one : kNat → kNat → Prop
  | eqe_left {a b c : kNat} : a =E b → kNat.rw_one b c
                                     → kNat.rw_one a c
  | eqe_right {a b c : kNat} : kNat.rw_one a b → b =E c
  …                                  → kNat.rw_one a c
  | sub_s {a b} : kNat.rw_one a b
                  → kNat.rw_one (kNat.s a) (kNat.s b)
  | sub_sum₀ {a₁ a b} : kNat.rw_one a b →
                        kNat.rw_one (kNat.sum a a₁) (kNat.sum b a₁)
  …
  | rl_cancel {n} : kNat.has_sort n MSort.NzNat →
                    kNat.rw_one (kNat.s n) n
```

In addition to basic definitions, some lemmas are also generated for various properties of the relations (unless omitted using the `with-lemmas` option). For instance, we have

```
@[refl] lemma eqe_refl (a : kBool) : a =E a := eqe.from_eqa eqa.refl
```

for the reflexivity of `=E` that follows from the similar property of `=A`. The lemma is prepended by the attribute `refl` that tells Lean to use this lemma with `refl`, `simp`, and other builtin tactics. More details about automatically-introduced lemmas are given in Section 5.1.

4.1. Proving the associativity and commutativity of the sum

With all these ingredients, we can prove properties of the `PEANO` module, like the associativity and commutativity of the sum. For the associativity law, instead of facing the general statement directly, we start with a first lemma, `sum_assoc_aux`, that takes an additional premise `o.ctor_only` telling that one of the operands `o` consists only of zero and successor symbols (because these have been marked with the `ctor` attribute in Section 2). Let us follow the proof thoughtfully.

```
lemma sum_assoc_aux (n m o : kNat) (oh : o.ctor_only) :
        n.sum (m.sum o) =E (n.sum m).sum o :=
begin
```

In an interactive proof in Lean, at every step of the proof text, we may have several goals, each one with a collection of premises or hypotheses. Tactics can be used to update these premises and goals in order to solve the latter. Usually, there is an active goal to

which the tactics are applied by default, and when all goals are solved the proof is done. For `sum_assoc_aux`, our initial proof state is

```
  { n : kNat, m : kNat, o : kNat, oh : o.ctor_only }
⊢ n.sum (m.sum o) =E (n.sum m).sum o
```

where the first line contains the premises, and the term after ⊢ is the current goal. We proceed by structural induction on the term `o` using the `induction` tactic. It will generate a goal for each case of the inductive definition of `kNat`, updating the premises and goal formula accordingly, and adding the appropriate induction hypotheses.

```
induction o,
```

In our case, three goals are generated for the three constructors of `kNat`, namely, `zero`, `s`, and `sum`. The `case` control statement in Lean allows focusing on the desired one.

```
case kNat.zero {                                        -- goal: n + (m + 0) =E (n + m) + 0
  simp,                                          -- applies the equations and the reflexivity of =E
},
```

For the `zero` base case, our proof state is

```
  { ... }
⊢ n.sum (m.sum kNat.zero) =E (n.sum m).sum kNat.zero
```

where we have omitted the irrelevant hypotheses. The goal can be reduced by the Lean simplifier to an identity, so invoking the `simp` tactic is enough to finish with this goal. This is because we have marked the `eq_` constructors of the *k*`.eqe` relations with the `simp` attribute to make the Maude equations visible to the simplifier.

```
case kNat.s : o ih {                                    -- goal: n + (m + o.s) =E (n + m) + o.s
  rw kNat.ctor_only at oh,
  simp [ih oh],
},
```

The inductive case of the successor is slightly more complex,

```
  { ih : o.ctor_only → n.sum (m.sum o) =E (n.sum m).sum o,
    oh : o.s.ctor_only, ... }
⊢ n.sum (m.sum kNat.zero) =E (n.sum m).sum kNat.zero
```

Our main assumptions are the induction hypothesis `ih`, and the argument `oh` of the lemma instantiated to the `s` constructor. The Lean simplifier can also solve the goal, but it needs the `=E` identity provided by the induction hypothesis. Its premise `o.ctor_only` can be obtained by rewriting the hypothesis `oh` with the definition of `ctor_only` using the `rw` tactic. Simplifying with the equations plus `ih oh` completes the proof of this case.

```
case kNat.sum : l r hl hr {
  rw kNat.ctor_only at oh,                          -- (l.sum .r).ctor_only evaluates to false
  contradiction,                                                  -- this case is not possible
},
end
```

In this latter case, we have

```
  { oh : (l.sum .r).ctor_only, ih : (does not matter), ... }
⊢ n.sum (m.sum (l.sum r)) =E (n.sum m).sum (l.sum r)
```

However, the induction hypothesis `ih` and the goal will be immaterial, since the premise `oh` is already false. In effect, `l.sum .r` is not a constructor term (because `sum` is not marked `ctor` in Maude). If we rewrite `oh : (l.sum .r).ctor_only` with the definition of `ctor_only`, we obtain `false` as a premise. The `contradiction` tactic spots this clash, solves the goal, and ends up the proof.

This lemma can be easily extended to arbitrary terms after the following sufficient-completeness result, whose proof is available in the repository [44].

```
lemma sc_nat (n : kNat) : ∃ w, n =E w ∧ w.ctor_only
```

Indeed, an arbitrary term `o` can be replaced by its equivalent constructor term `w` provided by `sc_nat`, and then `sum_assoc_aux` can be invoked to conclude.

```
theorem sum_assoc (n m o : kNat) : n.sum (m.sum o) =E (n.sum m).sum o
:= begin
  cases (sc_nat o) with w h,
```

In general, the `cases` tactic does a case distinction on the constructors of a term, introducing that many goals. In particular, it can be used to reveal an existential quantification, like the conclusion of the `sc_nat` lemma. This tactic introduces two new premises `w : kNat` (the element that exists) and `h : o =E w ∧ w.ctor_only` (the conditions it satisfies).

```
  simp [h.left],                                                    -- simplify with o =E w
```

Then, we simplify the goal with the identity `o =E w` (i.e. `h.left`) to yield `n.sum (m.sum w) =E (n.sum m).sum w`.

```
  exact sum_assoc_aux n m w h.right,
end
```

Notice we are already in the case of `sum_assoc_aux`, so we invoke that lemma with `n`, `m`, `w`, and `w.ctor_only` (i.e. `h.right`) as arguments. Its result is exactly the goal, so we conclude the proof with the `exact` tactic.

Similarly, we prove commutativity by first establishing the property for constructor terms, using two auxiliary lemmas, and then generalizing it through the sufficient completeness result `sc_nat`. Those auxiliary lemmas are versions of the two equations of the sum whose arguments are swapped, i.e., where commutativity is applied to these particular cases.

```
lemma zero_sum (n : kNat) (hc : n.ctor_only) : kNat.zero.sum n =E n :=
begin
  induction n,
  case kNat.zero { simp, },                                  -- 0 + 0 = 0 holds by the first equation
  case kNat.s : m hm {                                                -- the goal is sum 0 m.s = m.s
    rw kNat.ctor_only at hc,                                                -- get m is ctor_only
    simp [hm hc],                                        -- apply induction hypothesis, and implicitly
  },                                                                  -- the second ecuation and congruence
  case kNat.sum {
    rw kNat.ctor_only at hc,                                        -- is not ctor_only, so contradiction
    contradiction,
  },
end
```

The proof is similar to that of `sum_assoc_aux`, so we only comment the steps through the code. The counterpart for the `s` constructor is:

```
lemma s_sum (n m : kNat) (hc : m.ctor_only) : n.s.sum m =E (n.sum m).s
```

The omitted proof can be found in the repository [44]. Both lemmas are easily combined in the `sum_comm_aux` lemma, which plays the same role as the `sum_assoc_aux` for associativity.

```
lemma sum_comm_aux (n m : kNat) (hc : n.ctor_only) (hd : m.ctor_only)
  : n.sum m =E m.sum n :=
begin
  induction m,
  case kNat.zero { simp [zero_sum n hc], },                                 -- by lemma zero_sum above
  case kNat.s : o ih {
    rw kNat.ctor_only at hd,                                                        -- get m is ctor_only
    simp [ih hd],                                      -- equational reduction and hypothesis in the LHS
    simp [s_sum o n hc],                                                -- lemma s_sum above on the RHS
  },
  case kNat.sum {                                                               -- not a constructor term
    rw kNat.ctor_only at hd,
    contradiction,
  },
end
```

Finally, commutativity is obtained in general with the same argument used in `sum_assoc`.

```
lemma sum_comm (n m : kNat) : n.sum m =E m.sum n :=
begin
  -- get constructor terms v and w for n and m
  cases (sc_nat n) with v hv,
  cases (sc_nat m) with w hw,
  -- replace them in the proposition
  simp [hv.left, hw.left],
  -- apply sum_comm_aux
  exact sum_comm_aux v w hv.right hw.right,
end
```

5. Pragmatic extensions

The translation we have just presented is a faithful representation of the rewriting logic semantics in Lean, but we also want it to be a practical resource to prove interesting theorems about real Maude specifications. Anyone who has written a non-trivial proof with a

proof assistant knows that the simplest and most evident fact may require a bunch of definitions and auxiliary lemmas to get proved. In particular, when reasoning about specifications in rewriting logic, we usually assume that there is a single canonical form modulo axioms for any term, that we can reason inductively with constructor terms only without loss of generality, etc. However, these are facts (termination, confluence, sufficient completeness, etc.) that do not necessarily hold for an arbitrary specification and that cannot be easily checked in general (they are undecidable indeed). The Maude Formal Environment [17] includes termination [15], confluence [16], and sufficient completeness [22] checkers for trying to prove these properties, so we can trust their outcomes and state the required properties as axioms in the Lean specification. Of course, we can also prove them directly in Lean if we need them. Our tool does not currently try to prove these hard properties by itself neither it integrates the tools in the Maude Formal Environment, but some other very useful lemmas and optimization are provided to simplify the task of building a proof.

5.1. Lemmas

As anticipated in Section 4, the `maude2lean` tool introduces several lemmas in the translated Lean program to facilitate the proof task. For each kind, we generate four lemmas on the `=A` and `=E` relations, telling that `=E` is reflective and both are congruences.

```
lemma eqe_refl (a : k) : a =E a := eqe.from_eqa eqa.refl
lemma eqa_congr {a b c d : k} : a =A b → c =A d → (a =A c) = (b =A d)
lemma eqe_congr {a b c d : k} : a =E b → c =E d → (a =E c) = (b =E d)
lemma eqa_eqe_congr {a b c d : k} : a =A b → c =A d
                                  → (a =E c) = (b =E d)
```

The proof of `eqe_refl` concludes that `=E` is reflexive from the reflexivity of `=E` through the `eqe.from_eqa` axiom. For the other statements, we have introduced a generic lemma for congruences that we instantiate appropriately for each of them.

```
lemma generic_congr {α : Type} {rl ru : α → α → Prop}
  (cleft : ∀ {x y z}, rl x y → ru y z → ru x z)
  (cright : ∀ {x y z}, ru x y → rl y z → ru x z)
  (asymm : ∀ {x y}, rl x y → rl y x)
  {a₀ a₁ b₀ b₁ : α} : rl a₀ b₀ → rl a₁ b₁ → (ru a₀ a₁) = (ru b₀ b₁)
```

Moreover, for each pair of sorts $s < s'$ in the sort order, we generate a lemma `subsort_`s`_`s'

```
lemma subsort_s_s' (t : k) : t ⊳ s -> t ⊳ s'
```

telling that any term in the subsort is the supersort.

Regarding the rewriting relations, we also state four congruence lemmas for each kind k:

```
lemma eqe_rw_one_congr {a b c d : k} : a =E b → c =E d → (a =>1 c) = (b =>1 d)
lemma eqa_rw_one_congr {a b c d : k} : a =A b → c =A d → (a =>1 c) = (b =>1 d)
lemma eqe_rw_star_congr {a b c d : k} : a =E b → c =E d → (a =>* c) = (b =>* d)
lemma eqa_rw_star_congr {a b c d : k} : a =A b → c =A d → (a =>* c) = (b =>* d)
```

They are also proven using the `generic_congr` lemma. We also derive lemmas like

```
lemma rw_star_sub_s (a b : kNat) : a =>* b → s a =>* s b
  := by infer_sub_star ``(rw_one.sub_s) ``(eqe.eqe_s)
```

claiming that a derivation with `=>*` in an argument yields a derivation in the whole term. This follows by induction on the definitions of `=>1` and `=>*`, where `infer_sub_star` is a tactic we have programmed at the Lean metalevel to share this proof among their multiples instances. We have resorted to the metalevel because the statements of these lemmas involve the shape of the term and thus cannot be abstracted into a generic object-level result like the `generic_congr` lemma. Of course, we could have generated Lean code in our tool for every instance of the `rw_star_sub_`f lemmas, but this would have been less elegant and too verbose.

In general, constructors and lemmas for standard properties like reflexivity, symmetry, transitivity, and congruence are marked with the predefined Lean attributes `refl`, `symm`, `trans`, and `congr` that announce them to the automation and interactive tactics of the theorem prover. For instance, the Lean simplifier `simp` is able to automatically reduce Maude terms with equations and prove sort membership claims in most cases.

In Lean 4, the metaprogramming facilities used for `infer_sub_star` and some predefined attributes like `symm` and `trans` have been removed, replaced or moved to *mathlib*, so some helper results of our tool are not currently available for the new version of the theorem prover. When the replacement of these Lean features are enough documented, we will be able to incorporate them to the translation.

5.2. Maximum sort optimization

The translations of non-ground equations and rules include premises $x \rhd s$ requiring that the variable x in the statement belong to a specific sort s, as indicated in its Maude declaration. While the Lean simplifier may be able to deal with these premises, in some cases we will need to give proofs of them, which is tedious and sometimes unnecessary. Indeed, if we have a premise $x \rhd s$ and

- s is the maximum of the subsort relation in its kind $[s]$, and
- the kind $[s]$ does not contain error terms (i.e. partial functions),

we know that $x \rhd s$ trivially holds, and we can safely remove all such premises with s in the right-hand side from any statement in the module. This optimization in enabled with the `with-error-free-opt` flag of the configuration.

5.3. Derived operators as definitions

In algebraic specification, we usually distinguish between constructor and derived operators. Ideally, we want to take without loss of generality a constructor term as representative of any term, but this is only valid when the module is sufficiently complete [22]. While this property is undecidable in general, in some cases we can easily conclude that the equations defining an operator are complete, as most functional languages do.

With the `with-derived-as-def` option, the `maude2lean` tool will examine the equations of the derived symbols in a module to decide whether the symbol can be translated to an exhaustive Lean definition, instead of being included as a constructor of the inductive type for its kind. Conceptually, this is a fundamental change since the symbol will no longer belong to the signature of the kind (i.e. terms *will not* include that symbol), but become a function for generating a term at the Lean level. Indeed, we would have $(f\ t_1\ \dots\ t_n) = t'$ with the standard Lean equality, while we only have $(f\ t_1\ \dots\ t_n) =\mathbb{E}\ t'$ before. In practice, induction on this kind will not consider these derived operators, which may be helpful.[7]

Pattern matching in Lean is more restrictive than in Maude (all functions are required to be total, patterns must be linear, etc.). Thus, not every Maude operator can be translated along with its equations to a Lean definition. In order to translate a derived free symbol f, the following should be satisfied:

1. All equations defining it must be unconditional and not include the `owise` flag.
2. Their left-hand sides must be linear patterns, i.e. they must only contain constructor symbols and the same variable must not appear twice.
3. Patterns must cover all cases and be disjoint. In other words, any term with f on top must match the left-hand side of one and only one equation.

In particular, symbols whose arguments may contain error terms are not eligible to be translated to Lean definitions, because an exhaustive case distinction on them considering only constructor terms is not possible. The verbose flag `-v` of the tool activates some messages indicating whether each operator has been selected for this translation and explaining the reason why it has been discarded in such case.

While these requirements are motivated by the restrictions in Lean definitions and cannot be essentially weakened, we could partially drop the restriction on the `owise` attribute and the disjoint patterns, because cases in a Lean definitions are processed in order. However, this is not currently implemented.

For example, the `with-derived-as-def` option on the Peano numbers specification in Section 4.1 yields the following inductive definition for the kind `kNat`

```
inductive kNat
  | zero
  | s : kNat → kNat
```

where `sum` is missing, and the following definition for `sum`

```
def kNat.sum : kNat → kNat → kNat
  | n kNat.zero := n
  | n (kNat.s m) := (kNat.s (kNat.sum n m))
```

Other operators like `or`, `not`, and `implies` of the `Bool` kind have also been converted to Lean definitions. However, some others like `and` and `xor` have been excluded by the reasons we can see with the `-v` flag:

```
Info: the definition of _and_ is not convertible to Lean:
      pattern A:Bool and A:Bool is not linear.
```

5.4. Maude special operators

Some operators in Maude are said to be *special* since its behavior is not defined equationally in Maude, but primitively in the C++ implementation of the interpreter. They are recognized by the `special` attribute in their declarations. The main examples are the natural `Nat`, integer `Int`, and floating-point `Float` numbers, and strings `Strings`, whose operations are not defined by equations. Moreover, for `Float` and `String`, constants are atomic values that are not even built as terms from an explicit signature.

[7] Proofs where some arguments are assumed to be constructor terms are possible without any optimization using a premise with the `ctor_only` predicate, as shown in Section 4.1.

The pervasive polymorphic operators `_==_`, `_=/=_`, and `if_then_else_fi` are special too. Hence, reasoning about them in the translated Lean program would not be possible.

The approach to circumvent this problem is to insert artificial equations for the missing ones. Those for the `if_then_else_fi` operator are straightforward,

```
eq_itet {l r} : k.eqe (k.ifthenelsefi kBool.true l r) l
eq_itef {l r} : k.eqe (k.ifthenelsefi kBool.false l r) r
```

where `ifthenelsefi` are the translation of the operator name for Lean. Similarly, equations can be provided for the sum of natural and integer numbers. For example, if we use the `if_then_else_fi` polymorphic operators for the `Nat` type in the Peano example, we will obtain the following additional equations that are not present in the Maude source.

```
with kNat.eqe : kNat → kNat → Prop
  ...
  | eq_itet {l r} : kNat.eqe (kNat.ifthenelsefi kBool.true l r) l
  | eq_itef {l r} : kNat.eqe (kNat.ifthenelsefi kBool.false l r) r
```

The insertion of these noncontroversial equations is enabled by default.

However, the adaptation of other sorts like `Float` or `String` would be much more complex, and the approach explained in the next section seems more appropriate. Moreover, the translation of `_==_` is controversial. It can be translated to `=E` assuming that the module is admissible, but these may not always be true in practice. Hence, we do not introduce this equation automatically, and let the user do it manually if appropriate.

5.5. Native types

Some types in Maude have a natural counterpart in Lean. For example, Maude's `Bool` matches with Lean's `bool` (`Bool` in Lean 4), `Nat` with `nat`, `Int` with `int`, and so on. One possible simplification of the translation is to directly replace these Maude types by the corresponding Lean types. However, this may not be straightforward, the repertory of operations of both types may not coincide, and several problems may arise. For example, `Nat` and `Int` are sorts of the same kind in Maude, but they are completely different types in Lean.

Currently, the translation tool supports replacing the `Bool` sort by Lean's `bool` type with the option `with-native-bool`. The constants `true` and `false` are mapped to `tt` and `ff`, the common operators are translated to the Lean ones (`and` to `&&`, `or` to `||`, `not` to ¬, etc.), and additional operators only available in Maude are specified with artificial equations. Notice that there are some risks in this translation. For example, if the user introduces the equation `true = false`, we would have `true =E false` without a native translation, which is not directly inconsistent, but `tt = ff` with the native `bool`, which is directly inconsistent. The real problem is that this identity may also be indirectly and inadvertently introduced in a more discrete way.

For example, by activating the option `with-native-bool` in the running Peano module, no `kBool` type will be defined in Lean, the standard `bool` will be used instead of the former `kBool`, common constants and operators are translated as explained above, and those `Bool` operators without a Lean counterpart will be defined apart as constants,

```
namespace bool  -- Non-native operators
  constant xor : bool → bool → bool
  constant implies : bool → bool → bool
end bool
```

with their relations defined as axioms

```
namespace bool
  axiom eqe_xor {a₀ b₀ a₁ b₁ : bool} : a₀ = b₀ → a₁ = b₁
                                     → (bool.xor a₀ a₁) = (bool.xor b₀ b₁)
  axiom xor_comm {a b} : (bool.xor a b) = (bool.xor b a)
  axiom eq_implies {a b : bool} : (implies a b) = (¬ (xor a (a && b)))
  ...
end bool
```

6. Examples

In this section, we illustrate, using the dining philosophers example, how the translation allows practitioners to prove non-trivial properties from Maude specifications. Then, we summarize some examples that have been translated and analyzed using our tool.

6.1. The dining philosophers

We briefly describe in this section the proof in Lean by induction of a simple property of a Maude specification of the classical concurrency problem of the dining philosophers. It has been specified in Maude several times [18,37] and multiples properties have been proven or refuted by model checking [42]. However, classical model checking can only handle instances of the problem for

a fixed number of philosophers, so it is impossible to prove that a deadlock state is always reachable regardless of the number of messmates. Inductive reasoning can solve this problem easily, so we use an inductive proof in Lean to establish the property in the same specification we have already model checked.

The full specification of the problem is available and thoroughly explained in [42], so we only give a high-level overview here. In the initial configuration of the problem, generated by a Maude function `initial(n)`, there are n philosophers with empty hands `(o | k | o)` and n forks ψ between them in the circular table. Configuration terms are built from the following signature:

```
sorts Obj Phil Being List Table .
subsorts Obj Phil < Being < List .

op (_|_|_) : Obj Nat Obj -> Phil [ctor] .
ops o ψ : -> Obj [ctor] .
op empty : -> List [ctor] .
op __ : List List -> List [ctor assoc id: empty] .
op <_> : List -> Table [ctor] .

var L : List . var P : Phil .

eq < ψ L P > = < L P ψ > .
```

Every philosopher is a triple $(l \mid k \mid r)$ with an identifier k and two objects in their left l and right r hands. These objects can be either a fork ψ or nothing `o`. The dining table is a list of sort `List` wrapped in a `<_>` symbol that allows equations and rules to easily match the whole list. Circularity is achieved by considering that both ends of the list are connected, and the equation above normalizes the *circular* list by pushing the fork from the leftmost position to the rightmost one, which are virtually the same. The basic actions of the philosophers are described by means of rules

```
var Id : Nat . var X : Obj . var L : List .

rl [left] : ψ (o | Id | X) => (ψ | Id | X) .
rl [right] : (X | Id | o) ψ => (X | Id | ψ) .
rl [left] : < (o | Id | X) L ψ > => < (ψ | Id | X) L > .
rl [release] : (ψ | Id | ψ) => ψ (o | Id | o) ψ .
```

that make some philosopher take either fork or release them back to the table.

Our first goal is proving that from the initial term we can always reach a configuration where every philosopher has taken its right fork, as expressed below.

$$\begin{array}{rl} & < (o \mid 0 \mid o) \quad \psi \quad \cdots \quad \psi \quad (o \mid n-1 \mid o) \; \psi > \\ \rightarrow^* & < (o \mid 0 \mid \psi) \qquad \cdots \qquad (o \mid n-1 \mid \psi) > \end{array}$$

In Lean, we realize this as the `has_deadlock` theorem that can be proven by a standard induction on n and distinguishing cases on the structure of terms

```
theorem has_deadlock (n : ℕ) : ∃ t,
  (initial (nat_l2m n)) =>* t ∧ t.every_phil_has_right_fork
```

where `nat_l2m : ℕ → kNat` and `every_phil_has_right_fork : kTable → Prop` are Lean definitions to convert natural numbers from Lean to Maude

```
def nat_l2m : ℕ → kNat
  | 0            := kNat.zero
  | (nat.succ n) := kNat.s (nat_l2m n)
```

and to recognize the desired final state. However, we still need to prove that such a final state is actually deadlocked, i.e. that no rule rewrite can take place on it. This is proven in the following theorem in Lean

```
theorem ephrf_deadlock (a b : kTable)
  (hf : every_phil_has_right_fork a) : ¬ (a =>1 b)
```

and its counterpart for the `kList` kind, since a term of `kTable` consists in a wrapped `kList`. Their proofs essentially rely on the constructive definitions of the relations, since we derive this negative property by induction on the type `a =>1 b`. Let us follow the proof one step at a time:

```
begin
  intro h,                                    -- reductio ad absurdum, h : a =>1 b, goal is false
```

Since our goal `¬ (a =>1 b)` is equal by definition to `(a =>1 b) → false`, we may want to assume `a =>1 b` with `intro h` and proceed by *reductio ad absurdum*. Afterwards, our set of hypothesis contains `h : a =>1 b` and the original `hf : a.every_phil_has_right_fork`. Our new goal is `false`.

```
  induction h,                                    -- induction on h : every_phil_has_right_fork a -> false
```

The relation `=>1` is inductively defined, so we proceed by induction on `h` and consider every constructor in the definition of `=>1`. The first ones are `eqe_left` and `eqe_right` that allow replacing both sides of the rewriting step with an equivalent term modulo equations.

```
  case rw_one.eqe_left : _ _ hlm hmr _ ih {                    -- each case is a rw_one ctor
    exact ih ((ephrf_eclass hlm).mp hf),                       -- ephrf is invariant by eqe
  },
```

In this case, we have `hlm : l =E m`, `hmr : m =>1 r` (the premises of the `eqe_left` constructor), `ih : m.every_phil_has_right_fork → false` (the induction hypothesis), and `hf : l.every_phil_has_right_fork` (the instance of `hf` for this case). We can prove `false` using `ih`, but we have to prove first its premise from `hf`. The following auxiliary lemma, proven by induction on the relation `=E`, is combined with `hlm` and `hf` to do so.

```
lemma ephrf_eclass {a b : Maude.kList} (h : a =E b) :
  every_phil_has_right_fork a ↔ every_phil_has_right_fork b
```

The term `(ephrf_eclass hlm).mp hf` has type `m.every_phil_has_right_fork`, so `ih` applied on it yields `false`. The tactic `exact` followed by a term that is definitionally equal to the goal `false` concludes its proof.

```
  case rw_one.eqe_right : _ _ _ _ _ ih {                       -- already the ih
    exact ih hf,
  },
```

In this simpler case, our relevant hypotheses are `ih : l.every_phil_has_right_fork → false` and `hf : l.every_phil_has_right_fork`, among other irrelevant ones like `l =1 m` and `m =E r`. `ih hf` is definitionally equal to `false`, so we are done with this case.

```
  case rw_one.sub_table : _ _ ih {                             -- rewrite inside <_>
    exact kList.ephrf_deadlock _ _ hf ih,                      -- the counterpart for kList
  },
```

As explained in Section 3, the `sub_` constructors of the `rw_one` relation reflect that rewrite inside the argument of a term are rewrites of that term. In this case, `sub_table` tell that a rewrite inside the argument of type `kList` (sort `List` in Maude) of a `<_>` term applies to the whole term. Thus, we have to prove that `a =>1 b` is not possible with `a` and `b` being terms of type `kList` and `=>1` being `kList.rw_one` instead of `kTable.rw_one`. We use an auxiliary lemma `kList.ephrf_deadlock`, whose proof (ommited here for brevity) is also an induction on the constructors of `kList.rw_one`.

```
  case rw_one.rl_left {                                        -- rule left, which is applied on top
    simp [Maude.kTable.every_phil_has_right_fork, Maude.kList.every_phil_has_right_fork] at hf,
    contradiction,
  },
```

Constructors named `rl_` followed by a rule label represent single applications of a rule at the top level. In this case, `rl_left` applies the second rule labeled as `left`. Here, the relevant hypothesis is

```
  hf : (table ((Maude.kList.o.phil h_id h_x).join (h_l.join Maude.kList.fork))).every_phil_has_right_fork
```

Notice that the philosopher `h_id` (this is a name instroduced by Lean) does not have the right fork because it is on the table. If we simplify `hf` with the definition of `every_phil_has_right_fork`, we will obtain `false` as an hypothesis and so a contradiction. The `contradiction` hypothesis (or `exact hf`) can be used to conclude this case.

```
  case rw_one.sub_initial : _ _ ih {                           -- rewrite inside initial(n)
    exact kNat.no_step ih,                                     -- there are no rules for Nat
  },
```

Finally, we must only consider the axiom `sub_initial` that allows doing a rewrite inside the argument of `initial`, because `initial` is also a operator of the `Table` kind. However, this argument is of type `kNat` and there are no rewrite rules on `Nat` in the original Maude module, so there is no possible rewrite using `sub_initial`. This is proven with the auxiliary lemma `kNat.no_step`, which is invoked here taking the induction hypothesis as argument.

```
lemma no_step {n m : Maude.kNat} : ¬ (n =>1 m) := -- n =>1 m → false
begin
  intro h,
  induction h,
  all_goals { contradiction },
end
```

Description	Size (Maude)	Size (Lean)	Size (proof)
Ninjas	17	537	897
Peano	24	434	329
Philosophers	39	1057	393
Qlock	129	446	470
TAS	63	354	244

Fig. 1. Benchmarks.

The proof of this lemma follows the same pattern that `ephrf_deadlock`'s, but here all cases can be automatically solved by Lean using the `contradiction` tactic applied to all goals with `all_goals`.

Observe that we have extensively used induction on the rewriting and equational relations to prove a negative result. Hence, this theorem cannot be proven in any of the related tools mentioned in Section 7. The complete commented proofs are available in the repository of our tool [44].

6.2. Benchmarks

In this section we summarize some benchmarks we have utilized to assess the usefulness of the tool.[8] We have used the following examples:

- Ninjas [4] is a population protocol where we prove that consensus can be reached from any configuration and generate a counterexample showing that not every possible execution arrives to consensus.
- Peano presents a proof of the associativity and commutativity of the addition of Peano numbers. It also provides, as part of the previous proof, a proof of the sufficient completeness of the definition of sum. See Section 4.1 for more details.
- Philosophers refers to the example detailed in the previous section.
- Qlock is a mutual exclusion protocol. In particular, it is a variant of Dijkstra's binary semaphore. The proof of the mutual exclusion in Qlock has been used as a benchmark for multiple theorem provers, so achieving the proof using our translation gives us confidence in the applicability of the approach.
- TAS (Test & Set) is another mutual exclusion protocol, where we prove that the critical section has at most one process at the same time.

Fig. 1 presents the details of the benchmarks, where the first column identifies the example and the next columns depict the size of the files in lines of code. Although the examples are small/medium, we observe that the size of the translated Lean specification consists of two independent parts: a "fixed" part for definitions related to Maude but independent of the given specification and a specification-dependent part which grows, not only with the size of the specification, but also with its complexity in terms of equational attributes. Since the translation supports some optional helper extensions, enabling them would increase the size of the generated Lean code while reducing that of the handwritten proofs. As expected, the proofs depend on the complexity of the specification only in part, as shown in Qlock and TAS, which prove the same property for different specifications. However, the most important aspect for the size of the proof is the complexity of the property, which is independent of the translation. These facts make us confident of the usability of the translation for proving properties of Maude specifications.

7. Related work

In this section we briefly discuss related proposals to theorem proving for rewriting logic specifications. First, the Heterogeneous Tool Set (Hets) [30] integrates different logics by defining them as institutions and translations between them as comorphisms. In this way, systems can be defined in the most convenient logic and then translated in order to prove their properties in other formalisms supporting different types of theorem proving. Maude has been integrated into Hets [10], so some automated theorem provers for first-order logic and more powerful tools such as Isabelle/HOL [38] could be used. However, the translation generated some proof obligations when importations were required that made the approach difficult to use. Moreover, this integration has not been adapted to newer versions of Maude and Hets, so it does not support some features and theorem provers. Note that in our case the translation is not as general as the one in Hets because we do not intend to build a general framework where several formalisms are combined, but a particular translation between two systems. This allows us to use a simpler approach and focus on strategies to simplify the translation and the proofs. Furthermore, the translation from order-sorted specifications to many-sorted algebras for theorem proving has been explored in [23,29]. However, these translations follow a different approach by replicating the order-sorted structure in the types of the target specification, and they do not cover the rule rewriting relations.

Some theorem provers have also been implemented in this context. The Maude ITP [9] is an inductive theorem prover for Maude specifications that has been used to verify some properties of those systems. However, it only implements basic strategies, does not include decision procedures or simplification strategies, and does not take advantage of Maude features like unification and narrowing [7], making it useful for basic systems only. Currently, a sequel of this discontinued tool, called NuITP [14], is being

[8] The source code of all benchmarks is available at https://github.com/fadoss/maude2lean/tree/main/test.

developed to take advantage of these Maude features, but the other limitations still apply. The theorem prover in [39], although originally developed for CafeOBJ specifications [12], supports Maude specifications and provides automated mechanisms for inferring proofs. However, these mechanisms rely on a reduced set of basic commands and heuristics focusing on observational transitions systems [36], so it cannot be straightforwardly applied to any specification. In both cases, our translation provides three important advantages: (i) it supports reasoning about rewrite rules, not just equations; (ii) relations are specified constructively, letting users prove negative properties; and (iii) it integrates into a widespread proof assistant with a broad library of definitions and proofs.

Maude is integrated with different SMT solvers [7], which allows users to discharge proof obligations generated from Maude specifications. Narrowing [21,2,28] and its combination with SMT solving [20,24,40] let users represent an infinite number of initial states by a finite number of states with variables and an infinite state graph from an initial state by a finite state graph from another initial state with variables. This kind of analysis significantly expands the range of systems that can be verified in Maude. Moreover, *reachability logic* [45] and its associated tools can also be used to deductively reason about Maude programs. Although these automated proof methodologies are useful in practice they cannot deal with several properties that require more powerful techniques, hence it complements but does not replace the approach presented here.

8. Conclusions and future work

In this paper, we have described a translation from rewriting logic to the calculus of inductive constructions, implemented in a tool that produces programs for the Lean theorem prover from Maude specifications. Consequently, Lean can be used to reason and prove properties about Maude models, as we have shown with two small examples. The sort membership, equational, and rewriting relations have been specified inductively to allow for proving negative properties that would not follow from a purely axiomatic specification. Finally, lemmas and integration with tactics are provided to facilitate reasoning and automation. The contributions of this work are (i) support for reasoning about rule rewriting in addition to equations, (ii) a constructive specification of the relations that allows proving useful negative properties of the models like deadlock, and (iii) a practical translation to a well-known and community-active proof assistant with a wide library of reusable definitions and proofs. This translation can be easily adapted to other tools based on the same or a similar formalism, like Coq [3].

Our experience elaborating proofs with this tool suggests the importance of some properties that we usually take for granted when reasoning about Maude programs, but which are not necessarily true and which are not assumed by our tool. They are namely the existence of canonical forms (which follows from confluence and termination) and the sufficient completeness of the equations, so that we can always reason with constructor-only terms. Some of the optional optimizations we have implemented try to deal with these difficulties. While proving confluence or sufficient completeness from scratch in Lean can be difficult and tedious, there are tools for automated assessment of these properties that we leverage on, like those included in the Maude formal environment. With a positive answer of the external tool on the properties, we can introduce an axiom like

```
axiom sc_k : ∃ (canon : k -> k),
  (∀ {u v}, u =E v ↔ canon u = canon v) ∧ (∀ u, (canon u).ctor_only)
```

Deriving a Lean proof of the statement would be more interesting, but this is hard to achieve since those tools apply quite different techniques.

There are some paths for future extension of this work. For instance, our translation operates on the flattened version of a module, without leveraging on the modular nature of Maude specifications to reason compositionally in Lean. This means reusing the results about imported modules and instantiating the propositions obtained for parameterized ones. The translation can also be extended with more resources and lemmas to simplify the user task, including proofs of relevant properties of the data types in the standard prelude, and with support for other aspects of rewriting logic like parallel rewriting or rewriting strategies [13].

CRediT authorship contribution statement

Rubén Rubio: Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Investigation, Formal analysis, Conceptualization. **Adrián Riesco:** Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Investigation, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work is partially supported by the Spanish Agencia Estatal de Investigación under project ProCode (PID2019-108528RB-C22/AEI/10.13039/501100011033) and by Comunidad de Madrid under project BLOQUES-CM (S2018/TCS-4339), co-funded by EIE Funds of the European Union.

Appendix A. Translation options

Name	Type	Default	Description
`description`	string		Free-text description of the settings
`source`	string		Path to the input Maude file (or list of paths)
`module`	string		Name of the Maude module to be transformed (or where to evaluate the metamodule)
`metamodule`	string		Metarepresentation of the module to be transformed
`sort-renaming`	dictionary		Renaming for sort names
`kind-renaming`	dictionary		Renaming for kind names (any sort in the kind can be used as key)
`op-renaming`	dictionary		Renaming for operator names
`prefer-quotes`	Boolean	False	Whether to eliminate symbols in operator names or prefer quotes to maintain them
`use-notation`	[relation name]	[]	Relations where infix notation is used instead of standard alphanumeric names
`declare-notation`	[relation name]	[]	Relations where infix notation is declared (even if not used, use-notation implies this)
`with-ctor-predicate`	Boolean	True	Whether to define a ctor_only predicate that holds only on constructor terms
`with-error-free-opt`	Boolean	False	Whether to optimize error-free kinds with a most general sort by removing sort membership premises for it
`with-lemmas`	Boolean	True	Whether to include simple lemmas derived from the base specification
`with-aliases`	Boolean	True	Whether to define aliases for the constructors of the inductive relations
`with-repr`	Boolean	True	Whether to define a pretty printing function for the terms
`with-rules`	Boolean	True	Whether to include rewrite rules and rewriting relations
`with-frozen`	Boolean	True	Whether the frozen attribute of operators is obeyed by skipping argument rewriting axioms
`with-simp`	Boolean	True	Whether to label statements with the simp, symm, trans, and congr attribute
`with-axiom-simp`	Boolean	False	Whether to include structural axioms in the simplifier (only useful if with-simp)
`with-sort2kind`	Boolean	True	Whether to define the 'kind' function from sorts to their corresponding kind types
`with-derived-as-consts`	Boolean	False	Whether operators that are not constructors are translated as constants outside the inductive datatype
`with-derived-as-defs`	Boolean	False	Whether compatible non-constructor operators are translated to Lean definitions outside the inductive datatype
`with-native-bool`	Boolean	False	Replace Maude's Bool sort by Lean's bool
`split-eqe`	[string]	[]	Split the application of an equation on top as a subrelation of eqe for the given kinds
`has-sort-symbol`	string	⊳	Infix notation for the sort membership relation
`eqa-symbol`	string	`=A`	Infix notation for equality module axioms
`eqe-symbol`	string	`=E`	Infix notation for equality modulo equations
`rw-one-symbol`	string	`=>1`	Infix notation for the one-step rewriting relation
`rw-star-symbol`	string	`=>*`	Infix notation for the reflexive and transitive closure of the rewriting relation
`outermost-namespace`	string	Maude	Name of the outermost namespace (leave empty to omit that namespace)
`with-original-stmt`	Boolean	False	Include the original statement as a comment along with its translation
`lean-version`	integer (among `3`, `4`)	3	Lean version to generate code for

References

[1] J. Avigad, L. de Moura, S. Kong, Theorem proving in Lean v3.23.0, https://leanprover.github.io/theorem_proving_in_lean/, 2023.

[2] K. Bae, S. Escobar, J. Meseguer, Abstract logical model checking of infinite-state systems using narrowing, in: F. van Raamsdonk (Ed.), RTA 2013, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013, pp. 81–96.

[3] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions, Texts in Theoretical Computer Science, Springer, 2004.

[4] M. Blondin, J. Esparza, S. Jaax, A. Kučera, Black ninjas in the dark: formal analysis of population protocols, in: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, Association for Computing Machinery, New York, NY, USA, 2018, pp. 1–10.

[5] A. Bouhoula, J. Jouannaud, J. Meseguer, Specification and proof in membership equational logic, Theor. Comput. Sci. 236 (2000) 35–132, https://doi.org/10.1016/S0304-3975(99)00206-6.

[6] R. Bruni, J. Meseguer, Semantic foundations for generalized rewrite theories, Theor. Comput. Sci. 360 (2006) 386–414, https://doi.org/10.1016/j.tcs.2006.04.012.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, R. Rubio, C.L. Talcott, Maude manual v3.4, https://maude.lcc.uma.es/maude-manual, 2024.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C.L. Talcott (Eds.), All About Maude - a High-Performance Logical Framework, LNCS, vol. 4350, Springer, 2007.

[9] M. Clavel, M. Palomino, A. Riesco, Introducing the ITP tool: a tutorial, J. Univers. Comput. Sci. 12 (2006) 1618–1650, https://doi.org/10.3217/jucs-012-11-1618.

[10] M. Codescu, T. Mossakowski, A. Riesco, C. Maeder, Integrating Maude into Hets, in: M. Johnson, D. Pavlovic (Eds.), AMAST 2010, Springer, 2010, pp. 60–75.

[11] mathlib community, T., The Lean mathematical library, in: CPP 2020, ACM, 2020, pp. 367–381.

[12] R. Diaconescu, K. Futatsugi, Logical foundations of CafeOBJ, Theor. Comput. Sci. 285 (2002) 289–318, https://doi.org/10.1016/S0304-3975(01)00361-9.

[13] F. Durán, S. Eker, S. Escobar, N. Martí-Oliet, J. Meseguer, R. Rubio, C.L. Talcott, Programming and symbolic computation in Maude, J. Log. Algebraic Methods Program. 110 (2020), https://doi.org/10.1016/j.jlamp.2019.100497.

[14] F. Durán, S. Escobar, J. Meseguer, J. Sapiña, NuITP: an inductive theorem prover for equational program verification, in: A. Bruni, A. Momigliano (Eds.), PPDP 2024, ACM, 2024, in press.
[15] F. Durán, S. Lucas, J. Meseguer, MTT: the Maude termination tool, in: A. Armando, P. Baumgartner, G. Dowek (Eds.), IJCAR 2008, Springer, 2008, pp. 313–319.
[16] F. Durán, J. Meseguer, On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories, J. Log. Algebraic Methods Program. 81 (2012) 816–850, https://doi.org/10.1016/j.jlap.2011.12.004.
[17] F. Durán, C. Rocha, J.M. Álvarez, Tool interoperability in the Maude formal environment, in: A. Corradini, B. Klin, C. Cîrstea (Eds.), CALCO 2011, Springer, 2011, pp. 400–406.
[18] F. Durán, M. Roldán, A. Vallecillo, Invariant-driven strategies for Maude, Electron. Notes Theor. Comput. Sci. 124 (2005) 17–28, https://doi.org/10.1016/j.entcs.2004.11.018.
[19] S. Eker, J. Meseguer, A. Sridharanarayanan, The Maude LTL model checker, in: F. Gadducci, U. Montanari (Eds.), WRLA 2002, Elsevier, 2004, pp. 162–187.
[20] S. Escobar, R. López-Rueda, J. Sapiña, Symbolic analysis by using folding narrowing with irreducibility and SMT constraints, in: C. Artho, P.C. Ölveczky (Eds.), Proceedings of the 9th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems, FTSCS 2023, Cascais, Portugal, 22 October 2023, ACM, 2023, pp. 14–25.
[21] S. Escobar, J. Meseguer, Symbolic model checking of infinite-state systems using narrowing, in: F. Baader (Ed.), RTA 2007, Springer, 2007, pp. 153–168.
[22] J. Hendrix, J. Meseguer, H. Ohsaki, A sufficient completeness checker for linear order-sorted specifications modulo axioms, in: IJCAR 2006, Springer, 2006, pp. 151–155.
[23] L. Li, E.L. Gunter, A method to translate order-sorted algebras to many-sorted algebras, in: H. Cirstea, D. Sabel (Eds.), WPTE 2017, 2017, pp. 20–34.
[24] R. López-Rueda, S. Escobar, J. Sapiña, An efficient canonical narrowing implementation with irreducibility and SMT constraints for generic symbolic protocol analysis, J. Log. Algebraic Methods Program. 135 (2023) 100895, https://doi.org/10.1016/J.JLAMP.2023.100895.
[25] N. Martí-Oliet, M. Palomino, A. Verdejo, Rewriting logic bibliography by topic: 1990-2011, J. Log. Algebraic Methods Program. 81 (2012) 782–815, https://doi.org/10.1016/j.jlap.2012.06.001.
[26] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, Theor. Comput. Sci. 96 (1992) 73–155, https://doi.org/10.1016/0304-3975(92)90182-F.
[27] J. Meseguer, Twenty years of rewriting logic, J. Log. Algebraic Program. 81 (2012) 721–781, https://doi.org/10.1016/j.jlap.2012.06.003.
[28] J. Meseguer, Generalized rewrite theories, coherence completion, and symbolic methods, J. Log. Algebraic Methods Program. 110 (2020), https://doi.org/10.1016/J.JLAMP.2019.100483.
[29] J. Meseguer, S. Skeirik, Equational formulas and pattern operations in initial order-sorted algebras, Form. Asp. Comput. 29 (2017) 423–452, https://doi.org/10.1007/s00165-017-0415-5.
[30] T. Mossakowski, C. Maeder, K. Lüttich, The heterogeneous tool set, Hets, in: O. Grumberg, M. Huth (Eds.), TACAS 2007, Springer, 2007, pp. 519–522.
[31] L. de Moura, S. Ullrich, The Lean 4 theorem prover and programming language, in: CADE 2021, Springer, 2021, pp. 625–635.
[32] L.M. de Moura, S. Kong, J. Avigad, The Lean theorem prover, in: CADE 2015, Springer, 2015, pp. 378–388.
[33] C. Newcombe, Why Amazon chose TLA+, in: ABZ 2014, Springer, 2014, pp. 25–39.
[34] T. Nipkow, L.C. Paulson, M. Wenzel, Isabelle/HOL - a Proof Assistant for Higher-Order Logic, LNCS, vol. 2283, Springer, 2002.
[35] U. Norell, Dependently typed programming in Agda, in: A. Kennedy, A. Ahmed (Eds.), TLDI 2009, ACM, 2009, pp. 1–2.
[36] K. Ogata, K. Futatsugi, Proof scores in the OTS/CafeOBJ method, in: E. Najm, U. Nestmann, P. Stevens (Eds.), FMOODS 2003, Springer, 2003, pp. 170–184.
[37] P.C. Ölveczky, Teaching formal methods based on rewriting logic and Maude, in: J. Gibbons, J.N. Oliveira (Eds.), TFM 2009, Springer, 2009, pp. 20–38.
[38] L.C. Paulson, Isabelle: a Generic Theorem Prover, LNCS, vol. 828, Springer, 1994.
[39] A. Riesco, K. Ogata, An integrated tool set for verifying CafeOBJ specifications, J. Syst. Softw. 189 (2022) 111302, https://doi.org/10.1016/j.jss.2022.111302.
[40] C. Rocha, J. Meseguer, C.A. Muñoz, Rewriting modulo SMT and open system analysis, J. Log. Algebraic Methods Program. 86 (2017) 269–297, https://doi.org/10.1016/J.JLAMP.2016.10.001.
[41] R. Rubio, Maude as a library: an efficient all-purpose programming interface, in: K. Bae (Ed.), WRLA 2022, Springer, 2022, p. 14.
[42] R. Rubio, N. Martí-Oliet, I. Pita, A. Verdejo, Model checking strategy-controlled systems in rewriting logic, Autom. Softw. Eng. 29 (2022) 7, https://doi.org/10.1007/s10515-021-00307-9.
[43] R. Rubio, A. Riesco, Theorem proving for Maude specifications using Lean, in: A. Riesco, M. Zhang (Eds.), ICFEM 2022, Springer, 2022, pp. 263–280.
[44] R. Rubio, A. Riesco, Maude to Lean translator, https://github.com/fadoss/maude2lean, 2024.
[45] S. Skeirik, A. Stefanescu, J. Meseguer, A constructor-based reachability logic for rewrite theories, Fundam. Inform. 173 (2020) 315–382, https://doi.org/10.3233/FI-2020-1926.