NuITP

alpha 36

An Inductive Theorem Prover for Maude Equational Theories

F. Durán, S. Escobar, J. Meseguer, and J. Sapiña

March 2026

Contents

1 Introduction and Preliminaries

NuITP is an inductive theorem prover for Maude equational specifications that combines powerful state-of-the-art techniques such as narrowing, equality predicates, constructor variant unification, order-sorted congruence closure, ordered rewriting, strategy-based rewriting, and several others in order to reason about Maude equational programs. NuITP is based on the inductive inference system in [16]. Short presentations of the tool were made at the PPDP'24 and PROLE'25 conferences [5, 6]. See NuITP's web site for additional examples (`https://nuitp.webs.upv.es/`).

Since NuITP makes use of the latest advances in the Maude system, such as variant unification, narrowing, rewriting with strategies, and meta-interpreters, for a full compatibility, it is recommended to run NuITP on the latest available version of the Maude interpreter (last nightly build from `https://github.com/maude-lang/Maude/releases`). Otherwise, the requirements are those of the Maude's interpreter itself.

In this section, we explain some basic assumptions on the Maude functional modules that NuITP can reason about and then provide some guidelines on the first steps on the use of the tool.

1.1 Background on Order-Sorted Equational Logic

We assume familiarity with order-sorted equational logic and rewriting logic, and refer the reader to [12, 16] on these matters. We assume known the concepts of an order-sorted signature Σ with sort poset $(S,<)$ and the term Σ-algebras T_Σ and $T_\Sigma(X)$ for X an S-sorted set of variables. In this section some notation used in the rest of the document is introduced.

The least sort of a term t is denoted $ls(t)$. The subterm of term t at a term position p is denoted $t|_p$ and the term replacement of $t|_p$ by w at position p is denoted $t[w]_p$. Given an order-sorted substitution θ, its domain is $dom(\theta)$ and its range $ran(\theta)$. The application of a substitution θ to a term t is denoted $t\theta$. The order-sorted equational deduction relation is denoted $E \vdash u = v$ and its associated E-equality relation is denoted $=_E$. The term Σ-algebras modulo E are $T_{\Sigma/E}$ and $T_{\Sigma/E}(X)$ and we denote the equivalence class of a term t by $[t]_E$ (or just $[t]$). The *E-unifiers* of an equation $u = v$ are substitutions θ such that $u\theta =_E v\theta$.

Given an order-sorted equational theory $\mathcal{E} = (\Sigma, E \cup B)$, where B is a collection of associativity and/or commutativity and/or identity axioms and Σ is B-preregular,[1] we can associate to it a corresponding *rewrite theory* $\vec{\mathcal{E}} = (\Sigma, B, \vec{E})$ by orienting the equations E as left-to-right rewrite rules. The rewrite relation $t \to_{\vec{E},B} t'$ holds iff there is a subterm $t|_p$ of t, a rule $u \to v \textit{ if } \phi \in \vec{E}$ such that ϕ is a collection of equalities $t_1 = t'_1 \wedge \cdots \wedge t_n = t'_n$, and a substitution θ such that $u\theta =_B t|_p$, $t' = t[v\theta]_p$, and $\forall 1 \leqslant i \leqslant n$, $t_i\theta \downarrow_{\vec{E},B} t'_i\theta$, where the relation $\_ \downarrow_{\vec{E},B} \_$ is explained below. We denote the reflexive-transitive closure of $\to_{\vec{E},B}$ by $\to^*_{\vec{E},B}$. By definition, the relation $u \downarrow_{\vec{E},B} v$ holds on terms u,v iff there exist terms w,w' such that $u \to^*_{\vec{E},B} w =_B w' \; {}^*_{\vec{E},B}\!\!\leftarrow v$. The narrowing relation $t \overset{\alpha}{\leadsto}_{\vec{E},B} t'$ holds iff there exists a rule $u \to v \textit{ if } \phi$ in $\vec{E}$ such that ϕ is a collection of equalities $t_1 = t'_1 \wedge \cdots \wedge t_n = t'_n$ and a disjoint B-unifier θ such that $u\theta =_B t|_p\theta$, $\forall 1 \leqslant i \leqslant n$, $t_i\theta =_B t'_i\theta$, and $t' = (t[v]_p)\theta$.

The requirements on $\vec{\mathcal{E}}$ allowing us to reduce equational reasoning to rewriting are the following:

(i) for each rule $u \to v \textit{ if } \phi$ in $\vec{E}$, $vars(v) \cup vars(\phi) \subseteq vars(u)$;

(ii) *sort-decreasingness*: for each rule $u \to v \textit{ if } \phi$ in $\vec{E}$ and substitution θ we must have $ls(u\theta) \geqslant ls(v\theta)$;

(iii) *strict B-coherence*: if $t_1 \to_{\vec{E},B} t'_1$ and $t_1 =_B t_2$ then there exists $t_2 \to_{\vec{E},B} t'_2$ with $t'_1 =_B t'_2$;

(iv) *confluence* (resp. *ground confluence*) modulo B: for each term t (resp. ground term t) if $t \to^*_{\vec{E},B} v_1$ and $t \to^*_{\vec{E},B} v_2$, then there exist rewrite sequences $v_1 \to^*_{\vec{E},B} w_1$ and $v_2 \to^*_{\vec{E},B} w_2$ such that $w_1 =_B w_2$;

(v) *termination*: the relation $\to_{\vec{E},B}$ is well-founded (for $\vec{E}$ conditional, we require *operational termination* [11]).

If $\vec{\mathcal{E}}$ satisfies conditions (i)–(v) (resp. the same, but (iv) weakened to ground confluence modulo B), then it is called *convergent* (resp. *ground convergent*). The key point is that then, given a term (resp. ground term) t, all terminating rewrite sequences $t \to^*_{\vec{E},B} w$ end in a term w, denoted $t!_{\vec{\mathcal{E}}}$, that is unique

[1]For B any combination of associativity and/or commutativity that for any terms t,t' if $t =_B t'$ then $ls(t) = ls(t')$. If additional unit axioms U are involved, one further assumes that the terms t and t' are fully simplified by the oriented equations $\vec{U}$ modulo B. Maude automatically checks the B-preregularity of an equational theory $\mathcal{E}$.

up to B-equality, and its called t's *canonical form.* A key result when $\vec{\mathcal{E}}$ is ground convergent is that for any ground terms t,t' we have $t =_{E\cup B} t'$ iff $t!_{\vec{\mathcal{E}}} =_B t'!_{\vec{\mathcal{E}}}$.

A ground convergent rewrite theory $\vec{\mathcal{E}} = (\Sigma,B,\vec{E})$ is called *sufficiently complete* with respect to a subsignature Ω, whose operators are then called *constructors*, iff for each ground Σ-term t, $t!_{\vec{\mathcal{E}}} \in T_\Omega$. Furthermore, for $\vec{\mathcal{E}} = (\Sigma,B,\vec{E})$ sufficiently complete w.r.t. Ω, a ground convergent rewrite subtheory $(\Omega,B_\Omega,\vec{E}_\Omega) \subseteq (\Sigma,B,\vec{E})$ is called a *constructor subspecification* iff $T_{\mathcal{E}}|_\Omega \cong T_{\Omega/E_\Omega\cup B_\Omega}$. If $E_\Omega = \emptyset$, then Ω is called a signature of *free constructors modulo axioms* B_Ω.

Given an order-sorted equational theory $\mathcal{E} = (\Sigma, E\cup B)$, the notation $\vec{\mathcal{E}}_U^{=}$ combines three specific transformations. The arrow notation $\vec{\mathcal{E}}$ signifies that the defining equations in E are oriented as rewrite rules, from left to right, to be applied modulo the axioms B. The subscript U indicates that the theory $\vec{\mathcal{E}}$ has been transformed into an equivalent theory $\vec{\mathcal{E}}_U$ where the identity axioms are removed from B and replaced by explicit rewrite rules $\vec{U}$. Specifically, if $B = B_0 \cup U$ where U are the identity axioms, the theory $\mathcal{E}_U$ operates modulo B_0 and includes the necessary rules to simulate the identities. Finally, the superscript $=$ denotes the enrichment of the theory with equality predicates. This adds a binary operator $=_s$ for each top sort s in the signature, defined by rules in $\vec{\mathcal{E}}_U^{=}$ that effectively perform a structural equality check over constructors modulo B_0. For equational theories $\mathcal{E}$ with constructors Ω used as input theories for NuITP, the theory $\vec{\mathcal{E}}_U$ is required to have *free constructors* modulo B_{0_Ω}.[2]

1.2 Symbolic Methods Used in NuITP

For a detailed account of all the symbolic methods that NuITP exploits, the interested reader is referred to [16]. However, since they are crucial for the effectiveness of NuITP and awareness of them can be very helpful for NuITP users, an *impressionistic* summary of them is given here.

Equationally Defined Equality Predicates. They are automatically provided by the theory $\vec{\mathcal{E}}_U^{=}$ described above. This supports a quite strong form of *equational simplification* of formulas, since they are simplified not only by the equations $\vec{E}_U$ of $\vec{\mathcal{E}}_U$, but also by equations defining *equality predicates* for constructors. Viewing symbol _=_ itself as a commutative equality predicate symbol, and assuming that $\Omega = \{0,s\}$ are the constructors for Peano natural numbers, the equality predicate definitions automatically added to $\vec{\mathcal{E}}_U^{=}$ are: $(n = n) = \mathit{true}$, $(s(n) = 0) = \mathit{false}$, and $(s(n) = s(m)) = (n = m)$. In this way, we can simplify an equation such as $s(s(s(0))) = s(0)$, which cannot be further simplified by the equations for, say, addition and multiplication, to *false*. In NuITP this is supported by the **EPS** inference rule to be amply illustrated later and described formally in Section 4.5.1.

Narrowing. The already explained narrowing relation with equations $\vec{E}$ modulo axioms B can be exploited to *symbolically evaluate* in one step a subexpression of the form $f(\vec{u})$ in a formula, where f is not a constructor and the $\vec{u}$ are constructor terms. In NuITP, this is supported by the **NS** and **NI** inference rules to explained and illustrated in Sections 4.5.6 and 4.6.2.

Variant Unification. The input theory $\vec{\mathcal{E}}_U$ always has a subtheory $\vec{\mathcal{E}}_{1_U}$ that enjoys the *finite variant property* (FVP), and therefore has a *finitary constructor variant unification algorithm* [7, 17]. In particular, the subtheory (Ω,B_{0_Ω}) is FVP in a trivial way.[3] In Maude, the subtheory $\vec{\mathcal{E}}_{1_U}$ can be identified by those equations declared with the `variant` attribute. If no equation has the `variant` attribute, then $\vec{\mathcal{E}}_{1_U}$ becomes (Ω,B_{0_Ω}). In NuITP, simplification of formulas by variant unification is supported by the **CVUL** inference rule amply illustrated later and explained in Section 4.5.2.

Congruence Closure. As we will further explain in Section 1.6, NuITP proves goals of the form $\Gamma \to \Lambda$, where Γ is a conjunction of equalities. We often would like to apply *modus ponens* assuming Γ to prove Λ. But this can only be done if the formula has no variables. Otherwise, since the formula is *implicitly universally quantified*, this can only be done by first making our formula ground, $\overline{\Gamma} \to \overline{\Lambda}$, by turning its variables into Skolem constants, and then using *modus ponens*. But this is quite limited for automatic simplification purposes, since the ground equations $\overline{\Gamma}$ are arbitrary and therefore in general

[2]In the same way B is decomposed in B_0 and its unit axioms, that is, $B = B_0 \cup U$, the axioms in the constructor subtheory are decomposed as $B_\Omega = B_{0_\Omega} \cup U_\Omega$.

[3]For the case when B_{0_Ω} contains associativity without commutativity axioms this is nor really true; but Maude's unification algorithm is optimized to find a complete finite set of unifiers modulo associativity when it exists, and gives a warning when it cannot find such a complete set (see [2]). For NuITP, this means that if after giving a **CVUL** inference step when B_{0_Ω} contains associativity without commutativity axioms Maude gives a warning about incomplete unifiers, then that **CVUL** inference step becomes invalid and has to be undone. To the best of our knowledge no such warnings have yet appeared when using NuITP.

neither confluent nor terminating. The key advantage of *order-sorted congruence closure* modulo axioms B_0 [13] is that it automatically transforms the ground equations $\overline{\Gamma}$ into a *convergent* modulo B set of equivalent ground equations, which can be used together with the equations in $\vec{\mathcal{E}}_U^{=}$ and the executable induction hypotheses to try to simplify $\overline{\Lambda}$ to *true*. In NuITP, simplification of formulas by congruence closure is supported by the **ICC** inference rule explained in Section 4.5.8. **ICC** is actually stronger than just congruence closure, since it uses $\vec{\mathcal{E}}_U^{=}$ for simplification within the congruence closure process. In this way it can sometimes detect that the equations Γ are unsatisfiable, thus proving the formula in that case.

Ordered Rewriting. The formula simplification power of NuITP involves also the use of *executable* induction hypotheses as extra rewrite rules. This means that such hypotheses are *orientable* as (possibly conditional) rewrite rules that are RPO-terminating in the user-provided order $\succ$ (see Section 1.5). But some such hypotheses, for example an equation like $x+y=y+x$, can never be made executable this way. Nevertheless, an hypothesis like $x+y=y+x$ can be executed by *ordered rewriting*. The key idea is that the hypothesis $x+y=y+x$ *can* be applied to a subterm $u+v$ in a formula in a *terminating* way if $u+v\succ v+u$. Ordered rewriting (including conditional ordered rewriting) is supported by NuITP as part of its formula simplification process.

1.3 Assumptions on the Maude specifications

The Maude specifications currently handled by the NuITP tool are functional modules of the form **fmod** $(\Sigma,E\cup B)$ **endfm**, with E a set of ground convergent equations modulo B, where B is a set of axioms, which can be any combination of associativity (A), commutativity (C) and identity (U) axioms.

NuITP implements the theory described in [16], which, among other things, assumes an order-sorted equational theory $\mathcal{E}=(\Sigma,B,E)$ that can be decomposed by subtheory inclusions:

$$(\Omega,B_\Omega,\emptyset)\subseteq(\Sigma_1,B_1,E_1)\subseteq(\Sigma,B,E),$$

which, from left to right, respectively correspond to the constructor subtheory, a subtheory that has the Finite Variant Property (FVP) [4], and the original theory itself.

NuITP relies on the user to clearly specify these two subtheories by means of Maude's `ctor` declarations and `variant` equation attributes (see an example in Section 5.5 and [2, Chapter 14] for more information). Specifically, NuITP operates on the following assumptions. On the constructor subtheory:

- Constructor symbols are declared with the `ctor` attribute.
- (Σ,B,E) is sufficiently complete with respect to the subsignature of constructors Ω.
- There are no equations in E identifying constructor terms. That is, constructors are *free* modulo B_Ω.[4]

On the FVP subtheory:

- Equations in E_1 must be unconditional, and must be declared with the `variant` attribute.[5]
- E_1 is assumed to have the FVP modulo B_1 property. Establishing whether a theory has the FVP or not is a semi-decidable problem which, in case it holds, can be easily checked using Maude [2, Chapter 14.2].

The remaining equations in $E\backslash E_1$ can be conditional, but their conditions should not have any extra variables not present in their lefthand side. Furthermore, equations should *not* use any built-in features such as the `_==_` equality predicate or the `owise` attribute.

[4]A subtler point is that, when the user theory is internally transformed by NuITP to remove from B_Ω its unit (U) axioms, the signature Ω must remain free modulo the remaining A or C axioms. This can often be achieved by declaring the unit element in a supersort. For example, the following constructor specification of multisets remains free when the unit axiom is removed (and is then added as an equation `M:MSet,null = M:MSet`):
```
sorts MSet NeMSet .
subsort NeMSet < MSet .
op null :  -> MSet [ctor] .
op _,_ :  MSet MSet -> MSet [assoc comm id:  mt] .
op _,_ :  NeMSet NeMSet -> NeMSet [ctor assoc comm id:  mt] .
```
Note that multiset union has only been declared as constructor for the `NeMSet` sort, whereas the unit element `null` has been declared on the supersort `MSet`. This means that when NuITP internally removes the unit axiom and turns it into an equation `M,null = M`, with `M` a variable of sort `MSet`, this equation can *never* simplify constructor terms, which can only have sort `NeMSet`. Therefore, constructors remain free modulo AC when the U axiom is removed.

[5]See [2, Chapter 14.3] for more information on Maude's variant equations requirements.

1.4 Clauses and multiclauses

The formulas that can be shown to be inductive theorems of a given Maude functional module by NuITP are what we call *multiclauses*. A multiclause is a formula of the form

$$(w_1 = w'_1 \wedge \ldots \wedge w_n = w'_n) \rightarrow ((u^1_1 = v^1_1 \vee \ldots \vee u^1_{m_1} = v^1_{m_1}) \wedge \ldots \wedge (u^k_1 = v^k_1 \vee \ldots \vee u^k_{m_k} = v^k_{m_k}))$$

which condenses into a single formula k clauses having the same condition $(w_1 = w'_1 \wedge \ldots \wedge w_n = w'_n)$, namely, the k clauses:

$$(w_1 = w'_1 \wedge \ldots \wedge w_n = w'_n) \rightarrow (u^1_1 = v^1_1 \vee \ldots \vee u^1_{m_1} = v^1_{m_1})$$

$$\ldots$$

$$(w_1 = w'_1 \wedge \ldots \wedge w_n = w'_n) \rightarrow (u^k_1 = v^k_1 \vee \ldots \vee u^k_{m_k} = v^k_{m_k}).$$

A multiclause with no condition, i.e.,

$$(u^1_1 = v^1_1 \vee \ldots \vee u^1_{m_1} = v^1_{m_1}) \wedge \ldots \wedge (u^k_1 = v^k_1 \vee \ldots \vee u^k_{m_k} = v^k_{m_k})$$

is understood as having condition `true`, that is,

$$\mathit{true} \rightarrow (u^1_1 = v^1_1 \vee \ldots \vee u^1_{m_1} = v^1_{m_1}) \wedge \ldots \wedge (u^k_1 = v^k_1 \vee \ldots \vee u^k_{m_k} = v^k_{m_k})$$

Of course, the mathematical representation used above, and in some other places of this document, must be entered in a corresponding ASCII form. Mathematical symbols are represented in the usual form, e.g., $\rightarrow$ is written as `->`, $\wedge$ as `/\`, and $\vee$ as `\/`. All variables must be written in their typed form, that is, a variable N of sort *Nat* must be written as `N:Nat`. Since Maude's syntax for expressions is user-definable, the Maude parser may also need some additional parentheses in an expression to parse it correctly. It will complain otherwise.

For example, the multiclause

$$x > 0 = \mathit{true} \wedge y > 0 = \mathit{true} \rightarrow x + z > 0 = \mathit{true} \wedge (y * z > 0 = \mathit{true} \vee z = 0)$$

will be written as

```
((X:Nat > 0 = true) /\ (Y:Nat > 0 = true))
-> ((X:Nat + Z:Nat > 0 = true) /\ ((Y:Nat * Z:Nat > 0 = true) \/ (Z:Nat = 0)))
```

In the current version of the tool, commands must be entered without line breaks. It may be quite inconvenient some times, but the I/O is still quite basic. For example, to set the above formula as goal, we will write it in a single line:

```
NuITP> set goal ((X:Nat > 0 = true) /\ (Y:Nat > 0 = true)) -> ((X:Nat + Z:Nat > 0 ...
```

For presentation purposes, in this document we will rearrange the text and tabulate it, but be aware that the tool will complain if the input is incomplete when you hit the return key.

1.5 RPO termination order specified with the `rpo` attribute

By assumption, the equations E of an input module **fmod** $(\Sigma, E \cup B)$ **endfm** are ground convergent and therefore terminating modulo B. However, in the process of developing an inductive proof of some property about such a module, new *induction hypotheses* are often added to the module. Some of these hypotheses may be *executable* as, perhaps conditional, rewrite rules, say, $\vec{H}_{exec}$. However, if the combined set of rules $\vec{E} \cup \vec{H}_{exec}$ is non-terminating, NuITP could loop, a trap that should be avoided in automated reasoning and, in particular, in inductive theorem proving. This trap can be avoided by making explicit a suitable reduction path order (RPO) relation $\prec$ [1] under which the module's original equations E are terminating (modulo the axioms[6] B). NuITP can then use this RPO order $\prec$ to automatically identify and orient a subset of executable hypotheses $\vec{H}_{exec}$ that are also RPO-terminating under the same order, thus avoiding non-termination. Furthermore, by making $\prec$ a *total* order on function symbols, two ground terms that are

[6]Such axioms should not include identity axioms. This is ensured by NuITP by means of an internal semantics-preserving theory transformation that transforms identity axioms into rules and also transforms the equations E to match without identity axioms.

different modulo B can always be compared under the $\prec$ order, which is very useful for some of NuITP's inference rules. To achieve these termination properties, NuITP requires that the user specifies a suitable RPO order modulo axioms that is total on function symbols by *ordering* the function symbols (including constants) in the signature of the input theory by means of a *tagging* of each of its operators with a natural number using the `rpo` attribute of Maude (see examples in Section 4.5.2 of [2]), so that, say, operator f is bigger than operator g iff f's number is bigger than g's number. This should be done so that all subsort-oveloaded version of each operators are annotated with the same number, and different operator symbols are annotated with different numbers. The way to do this is illustrated in the following example.

Consider the following equational theory in which no RPO order has been specified:

```
fmod PEANO+ADD-NO-ORDER is
  sorts Nat NzNat .
  subsorts NzNat < Nat .

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

  op _+_ : Nat Nat -> Nat [assoc comm] .
  eq N:Nat + 0 = N:Nat .
  eq N:Nat + s M:Nat = s(N:Nat + M:Nat) .
endfm
```

where `0` and `s_` are constructor symbols and `_+_` is a defined function symbol (defined by recursive equations). Then, a suitable RPO order for this theory making it terminating is `0` $\prec$ `s_` $\prec$ `_+_`. Thus, to declare this RPO order, starting by the *smaller* symbol in this order, operators must be tagged as follows:

```
fmod PEANO+ADD-WITH-ORDER is
  sorts Nat NzNat .
  subsorts NzNat < Nat .

  op 0 : -> Nat [ctor rpo 1] .
  op s_ : Nat -> NzNat [ctor rpo 2] .

  op _+_ : Nat Nat -> Nat [assoc comm rpo 3] .
  eq N:Nat + 0 = N:Nat .
  eq N:Nat + s M:Nat = s(N:Nat + M:Nat) .
endfm
```

Since we want to specify an RPO order to ensure termination of the equations defining the module's functions, any constructor symbol should be annotated with a smaller number than that of any defined symbol. Furthermore, to ensure that the RPO order is *total* on B-equivalence classes of ground terms (which is needed for some NuITP commands), it should never be the case that two syntactically different (not subsort-overloaded) operators are specified with the same `rpo` number declaring their RPO priority. Finally, in the above example we have used 1 as the starting index, but there is actually no restriction on the choice of the smallest value, provided that it is a natural number and that the intended order between symbols is preserved.

NuITP can then check the theory's RPO-termination modulo its axioms for the specified order using its check rpo command.

1.6 The Inductive Inference System behind NuITP

NuITP is based on the inductive inference system developed in [16]. The inference rules of the system transform inductive goals of the form $[\overline{X},\mathcal{E},H] \Vdash \Gamma \rightarrow \Lambda$ into sets of goals of the same form. In these goals, $[\overline{X},\mathcal{E},H]$ is an *inductive theory* with $\mathcal{E} = (\Sigma, E \cup B)$ a ground convergent and sufficiently complete equational theory (the program to verify), H the current induction hypotheses, and $\overline{X}$ a set of Skolem constants used in H and $\Gamma \rightarrow \Lambda$; and where $\Gamma \rightarrow \Lambda$ is a $\Sigma(\overline{X})$-multiclause, with $\Sigma(\overline{X}) =_{def} \Sigma \uplus \overline{X}$. The empty set of goals is denoted $\top$. This symbol will be used to mark a goal as *closed*, or proved, since leads to the empty set of goals. An initial goal will be represented as $[\varnothing,\mathcal{E},\varnothing] \Vdash \Gamma \rightarrow \Lambda$, with $\Gamma \rightarrow \Lambda$ a Σ-multiclause.

A proof tree is a tree of goals, with the initial goal that we want to prove at the root of the tree, and where the children of each node in the tree have been obtained by applying an inference rule in the usual bottom-up proof search fashion. Goals in the leaves are called the *pending goals*. A proof tree is *closed* if it has no pending goals, i.e., if all its leaves are marked with $\top$. The soundness of the inference system, proved in [16], means that if the goal $[\overline{X},\mathcal{E},H] \Vdash \phi$ is the root of a closed proof tree, then ϕ is valid in the inductive theory $[\overline{X},\mathcal{E},H]$. To increase its effectiveness, the inference system maintains the invariant that the induction hypotheses H in all inductive theories will always be in simplified form.

The tool provides basic functionality for managing the proof tree, including commands to show specific goals, the open goals, etc., to apply the inference rules of the inductive inference system in [16], and for undoing part of the proof. The tool can also generate files including the log of the proof or the state of the prover, which can then be re-entered to redo the proof or to recover the state, respectively. The tool can also generate LaTeX and HTML files documenting the state of the prover (see Section 4.1.3). As part of this documentation proof trees are generated, which are helpful to get a better idea of our proofs.

Goal simplification rules are easily amenable to automation, and NuITP provides mechanisms to define and apply tactics or strategies automating their application (see Sections 2.3, 4.3 and 3). Inductive rules are typically applied under user guidance, although they could also be automated by tactics.

1.7 Generator sets

With generator set induction, we can induct on a variable of sort s using not only the constructors of sort s (the so-called structural induction), but also any generator set, that is, any set of constructor tems that generate all the ground constructor terms of sort s as instances. Generator sets are particularly useful when constructors obey structural axioms B, including associativity or associativity-commutativity, for which structural induction may be ill-suited, leading to longer proofs. Generator sets may also be useful even for free constructors, mainly when trying to induct on a variable under a function equationally defined in a manner other than by standard primitive recursion.

Thus, a generator set for a sort s is just a set of constructor terms patterns of sort s or smaller such that, up to B-equality, any ground constructor term of sort S is a ground substitution instance of one of the patterns in the generator set. For Ω an order-sorted signature of constructors satisfying axioms B, decomposable as $B = B_0 \uplus U$, and S a sort in Ω, a B_0-*generator set* for sort S is a finite set of terms $\{u_1,\ldots,u_k\}$, with $u_1,\ldots,u_k \in T_\Omega(X)_s$ and such that

$$T_{\Omega/B_0,s} = \{[u_i\,\rho] \in T_{\Omega/B_0,s} \mid 1 \leqslant i \leqslant k,\ \rho \in [X \to T_\Omega]\}.$$

For example, structural induction on the Peano natural numbers, as defined, e.g., in the modules in Section 1.5, is achieved by the generator set {`0`, `s(N)`} for sort `Nat`, but the alternative generator set {`0`, `s(0)`, `s(s(N))`} may be better suited to reason about a function such as, for example, addition defined by the equations:

```
  eq X + 0 = X .
  eq X + s(0) = s(X) .
  eq X + s(s(Y)) = s(s(X + Y)) .
```

As we will see in the coming sections, the above generator sets will be written in NuITP as `0 ;; s(N:Nat)` and `0 ;; s(0) ;; s(s(N:Nat))`, respectively.

```
  NuITP> genset GEN-NAT for Nat is 0 ;; s N:Nat .

  Generator set GEN-NAT for sort Nat added.

  GEN-NAT (default):
    0
    s N:Nat

NuITP> genset GEN-NAT2 for Nat is 0 ;; s 0 ;; s s N:Nat .

  Generator set GEN-NAT2 for sort Nat added.

  GEN-NAT2:
    0
    s 0
    s s N:Nat
```

The usefulness of generator sets is even greater for constructors that satisfies structural axioms such as associativity or associativity-commutativity. For example, consider the following module, which specify sorts `NeMSet` and `MSet` of non-empty multisets and multisets, respectively, with `empty` for empty multisets and `_U_` for union of multisets.

```
fmod MSET is
  sorts NzNat Nat NeMSet MSet MBool .
  subsorts NzNat < Nat < NeMSet < MSet .
  vars NN NM : NzNat .
  vars N M : Nat .
  var  S : MSet .
  op 0 : -> Nat [ctor] .
  op s_ : Nat -> NzNat [ctor] .
  ops true false : -> MBool [ctor] .
  op _=e=_ : Nat Nat -> MBool [comm] .
  eq 0 =e= NN = false .
  eq s N =e= s M = N =e= M .
  op empty : -> MSet [ctor] .
  op _U_ : NeMSet NeMSet -> NeMSet [ctor assoc comm id: empty] .
  op _U_ : MSet MSet -> MSet [assoc comm id: empty] .
  op _in_ : Nat MSet -> Bool .
  eq N in empty = false .
  eq N in N = true .
  ceq N in M U S = true if N =e= M = true .
  ceq N in M U S = N in S if N =e= M = false .
endfm
```

The sets `0 ;; s N:Nat` and `empty ;; X:Nat ;; S1:NeMSet U S2:NeMSet`, corresponding to structural induction, are possible generator sets for sorts `Nat` and `MSet`, respectively. However, the generator set `empty ;; X:Nat ;; X:Nat U S:NeMSet` is much better suited to reason about the multiset membership predicate `_in_`.

As pointed out in Section 1.3, the signature of constructors Ω must be free modulo the A or C axioms, and indeed must remain so after the elimination of the identity attributes. Identity attributes may lead to technical complications when reasoning inductively about proper constructor subterms of a term. Thus, what was said about constructors is also applied to generator sets. Decomposing the set B of structural axioms as as a disjoint union $B = B_0 \uplus U$, with B_0 the associative and/or commutative axioms and U the unit (identity) axioms, we must consider generator sets modulo B_0, instead of modulo B. In the above specification, we may consider `empty ;; N:Nat U S:MSet` as a possible generator set modulo ACU for sort `MSet`. However, a term like `N:Nat U empty` *collapses* modulo B to its proper subterm `N:Nat`, and is, for that reason, problematic for inductive reasoning on *subterms*.

In practice, one may use *several* generator sets for the same sort S, depending on the various functions that one may wish to reason inductively about. We will see in the coming sections how we can define in NuITP several generator sets for each sort, and choose the most convenient one for each inference step. See Section 4.2 for details on how to define new generator sets and how to set a specific one as default. A simple method to check the correctness of generator sets is explained in [16, §2.9].

2 Getting started

NuITP runs with the latest version of the Maude System (last nightly build from https://github.com/maude-lang/Maude/releases).[7]

2.1 Running NuITP

Three key ideas to start using the tool:

- The current version of NuITP is distributed as a Maude file, named `NuITP.maude`.

[7]Note that NuITP makes use of some functions declared in the `file.maude` file, so you need to have the `MAUDE_LIB` environment variable declared and pointing to the folder where that file, together with the prelude and the rest of the default Maude System files, are located. Alternatively, you can have these files where the Maude binary is located, or load it manually like any other Maude file.

- To be able to read from and write into files,[8] Maude requires to be run with the `allow-files` or `trust` flags on (see the Maude manual [2, Chapter 9]). This flag is optional, but if permission is not granted the input/output facilities will not work.
- We can work on modules previously loaded in Maude, and once NuITP has been started, it is not possible to enter new modules on which to operate. Therefore, these modules need to be loaded in Maude before NuITP is started,

In summary, to run the tool, execute your Maude binary (with the `allow-files` or `trust` flag if want to use the tool's I/O facilities), load the files containing the modules to work with, and then load the `NuITP.maude` file.

```
$ maude -allow-files
              \||||||||||||||||||/
              --- Welcome to Maude ---
              /||||||||||||||||||\
          Maude git-eafdb6 built: Feb 19 2026 20:00:00
          Copyright 1997-2025 SRI International
              Mon Mar  9 00:13:49 2026

NuITP> load examples/peano+R.maude
NuITP> load NuITP.maude

    ==================================
                  NuITP
          Inductive Theorem Prover
       for Maude Equational Theories
       (alpha 36 built Mar 8th 2026)
    ==================================
            Copyright 2021-2026
    Universitat Politècnica de València
    ==================================

NuITP>
```

As any other Maude files, instead of loading them manually by means of Maude's `load` command, one can provide the files to load as arguments when starting the Maude system. Thus, one can get the exact same effect with:

```
$ maude -allow-files examples/peano+R.maude NuITP.maude
```

NuITP is an interactive tool, with a number of commands that you can use to interact with it, by typing them at the `NuITP>` prompt. The first two commands that you will need to use are:

`quit` (or `q` in its abbreviated form) to leave the tool,[9] and

`help` to get a basic help on the syntax on available commands.[10]

Once the NuITP tool is ready to accept commands, we can begin our interaction. We can carry out inductive proofs on any module *previously* loaded in Maude, but we need to tell the theorem prover which among these modules is the chosen one. To select a module we can use the `set module` command.

`set module` *<module_name>* `.`

For example,

[8] See Section 4.1.3 for details on how to load NuITP scripts, or save proof logs and reports in HTML and LaTeX formats.

[9] Note that with the `quit` command we leave NuITP and the Maude prompt appears. We must use a second `quit` command to also leave Maude.

[10] Most of NuITP's commands follow the Maude convention of ending with a dot. In NuITP, like in Maude, there are a few exceptions, including the `q`/`quit`, `help`, `load`, and `save`/`snap`/`texhtml` commands.

```
NuITP> set module PEANO+R .

  Module PEANO+R is now active.

NuITP>
```

Any proof in NuITP will have a top goal and a number of subgoals derived from it through the successive application of different proof commands. At any time, a proof is either completed, if there are no further goals to be proven, or open, if there are a number of open goals in what we call the *frontier* of the proof.[11] The following commands are useful to manage these goals:

`set goal` *<goal>* `.`
: Sets the specified goal as the active top goal. There can only be one top goal, therefore, setting a new top goal results in the deletion of the previous proof.

`show module .`
: Shows the currently active module.

`show goals .`
: Shows all goals, that is, the entire proof tree.

`show goal` *<goal_id>* `.`
: Shows the goal with the specified identifier.

`show frontier .`
: Shows the goals in the frontier, that is, the goals pending to be proved to complete the proof of the current top goal.

`show log .`
: Shows a raw log of the session.

The following commands are also available to load and save proof scripts and proof reports (note that these commands do not have a final dot, and can only be used if the appropriate permissions, with the `allow files` or `trust` flags, were granted):

`load` *<file-name>*: loads (and executes) a proof script or restores a NuITP session snapshot.

`save` *<file-name>*: saves the current proof script in the specified file.

`snap` *<file-name>*: saves a snapshot of the current NuITP session in the specified file.

`tex` *<file-name>*: saves a LaTeX report of the current proof in the specified file.

`html` *<file-name>*: saves a report of the current proof in HTML5 format in the specified file.

Loading additional modules without leaving Maude You can skip the rest of this section and start using the tool. However, advance users may welcome some further information on how to interact with the tool. NuITP is written in Maude, and ts top module, in the `NuITP.maude` file, is named `NuITP`. Since we can only work on modules previously loaded in Maude, once NuITP has started, it is not possible to enter new modules on which to operate. This is the reason why for basic uses we explained above that these modules are to be loaded in Maude before NuITP is loaded (and automatically started. However, if we need to, we could load additional modules without leaving Maude. To do that, we must leave NuITP (perhaps saving the state of our proof or a report of it), load the new modules, and then re-start NuITP. The reason why the tool is started when the `NuITP.maude` file is loaded is simply because the file ends with a `erew init` command. Thus, if the tool is stopped, for example by a `q` commnad or with a control-C, you may be able to reinitiate it without leaving Maude. To start it you just need to type

```
Maude> erew init .
```

with the `NuITP` module selected, or directly

[11] If we view the goal-subgoal relation as a binary tree rooted at the top goal, the "frontier" corresponds to the leaves of that tree, i.e., to the set of currently unproved (sub)goals, whose proof is necessary to prove the top goal.

```
Maude> erew in NuITP : init .
```

2.2 A simple proof: associativity of addition

In this section, we illustrate the basic operation of the theorem prover. As already stated, in addition to the commands related to the management of the proof tree itself, the prover provides several commands, which basically apply corresponding inference rules. Please, see [16] for detailed descriptions of the inference rules, and Section 4 for a detailed information, concrete syntax, and examples on each command of the tool. Some additional examples can be found in Section 5.

All these commands have the form

`apply` *<inference_rule_name>* `to` *<goal_id> [<possible_additional_arguments>]* `.`

which basically apply a given inference rule to a specific goal. For example, we can request the application of the Equational Predicate Simplification (`EPS`) rule on a specific pending goal, say `0.1`, by giving the command:

```
NuITP> apply eps to 0.1 .
```

Let us see some of these commands in action. We begin with something simple like proving associativity of natural number addition using its definition in Peano notation in the following module:

```
set include BOOL off .

fmod PEANO+R is
  sorts Nat NzNat .
  subsorts NzNat < Nat .

  op 0 : -> Nat [ctor rpo 1] .
  op s_ : Nat -> NzNat [ctor rpo 2] .

  op _+_ : Nat Nat -> Nat [rpo 3] .
  eq N:Nat + 0 = N:Nat .
  eq N:Nat + s M:Nat = s(N:Nat + M:Nat) .
endfm
```

The module `PEANO+R` defines sorts `Nat` and `NzNat` with `0` and `s_` in the usual way. It also defines a `_+_` operation. The `rpo` attributes define an RPO order that makes the module's equations terminating and that will be used to orient future induction hypotheses as rewrite rules when possible. The syntax used to define RPOs is taken directly from the MTA tool [9]. See Section 1.5 for a brief presentation of RPOs and the syntax introduced by the MTA and used here.

By default, the `BOOL` module is imported in all modules. With the "`set include BOOL off .`" Maude command we force Maude to deactivate such an inclusion before loading our file. Maude's "`set include BOOL off .`" command should be given before any module that is entered into NuITP. This is because of the non-built-in features of Maude added by the BOOL module. However, if the user needs Boolean values, the module `TRUH-VALUE` imports just the truth values without creating any built-in-related problems. Likewise, the `BOOL-OPS` module imports not just the truth values, but also all the usual Boolean operations without any built-in related problems.

To begin our proof, we first need to load the `peano+R.maude` file and then start the tool.

```
$ maude -allow-files examples/peano+R.maude NuITP.maude

         \||||||||||||||||||/
       --- Welcome to Maude ---
         /||||||||||||||||||\
 Maude git-eafdb6 built: Feb 19 2026 20:00:00
  Copyright 1997-2026 SRI International
       Mon Mar  9 00:13:49 2026
```

```
    ==================================
                  NuITP
         Inductive Theorem Prover
      for Maude Equational Theories
      (alpha 36 built Mar 8th 2026)
    ==================================
            Copyright 2021-2026
    Universitat Politècnica de València
    ==================================

NuITP>
```

Then, we set the module as the current active one. Please, note that the tool assumes that the module is executable (it's terminating, using the specified RPO, it's ground confluent, and sufficiently complete). It is the responsibility of the user to prove these properties.

```
NuITP> set module PEANO+R .

  Module PEANO+R is now active.

NuITP> genset GEN-NAT for Nat is 0 ;; s N:Nat .

  Generator set GEN-NAT for sort Nat added.

  GEN-NAT (default):
    0
    s N:Nat
```

Then, we set the goal corresponding to the associativity of the `_+_` operator.

```
NuITP> set goal X:Nat + (Y:Nat + Z:Nat) = (X:Nat + Y:Nat) + Z:Nat .

  Initial goal set.

  Goal Id: 0
  Generated By: init
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $1:Nat + ($2:Nat + $3:Nat) = ($1:Nat + $2:Nat) + $3:Nat
```

One first observation on the `set goal` command is that variables are internally renamed. Commands acting on specific variables must refer to them using their new names.

Once the top goal has been set, with identifier `0`, we can start our proof. In this case, the thing to do is to apply generator-set induction (`GSI`) on one of the variables. For example, we can apply it on the variable `$3:Nat`, using the generator set given by (i) `0` and (ii) `s` *N* for *N* a natural value. Note that the generator terms in such a set are separated by `;;`. Note also that the generator set `0 ;; s(K:Nat)` corresponds to the standard induction on the natural numbers.[12]

```
NuITP> apply gsi to 0 on $3 with 0 ;; s(K:Nat) .

  Generator Set Induction (GSI) applied to goal 0.
```

[12]Generator sets do not need to be explicitly given in every command requiring one. See Section 4.2 for information on how to define and use generator sets for the different sorts in your specifications.

```
  Goal Id: 0.1
  Generated By: GSI
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $1:Nat + ($2:Nat + 0) = ($1:Nat + $2:Nat) + 0

  Goal Id: 0.2
  Generated By: GSI
  Skolem Ops:
    $4.Nat
  Executable Hypotheses:
    ($1:Nat + $2:Nat) + $4 => $1:Nat + ($2:Nat + $4)
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $1:Nat + ($2:Nat + s $4) = ($1:Nat + $2:Nat) + s $4
```

Two new goals have been created, with identifiers `0.1` and `0.2`, which define the current frontier of the proof. They can be easily discharged by equality predicate simplification `EPS` as follows.

```
NuITP> apply eps to 0.1 .

  Equality Predicate Simplification (EPS) applied to goal 0.1.

  Goal 0.1.1 has been proved.

  Unproven goals:

  Goal Id: 0.2
  Generated By: GSI
  Skolem Ops:
    $4.Nat
  Executable Hypotheses:
    ($1:Nat + $2:Nat) + $4 => $1:Nat + ($2:Nat + $4)
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $1:Nat + ($2:Nat + s $4) = ($1:Nat + $2:Nat) + s $4

  Total unproven goals: 1
```

Note that the tool proves the goal, and shows the remaining, unproved goals. We can finish the proof by proving goal `0.2`, also by equality predicate simplification.

```
NuITP> apply eps to 0.2 .

  Equality Predicate Simplification (EPS) applied to goal 0.2.

  Goal 0.2.1 has been proved.
  qed
```

When there are no pending goals, the tool will show the classical `qed` symbol (*quod erat demonstrandum*), to inform us about such a fact.

2.3 An alternative proof with ! commands: associativity of addition

The equational simplification of goals after the application of some other inference rules is quite effective. Indeed, it is so common to apply it after other commands that NuITP provides *modified* versions of some of its commands, including `gsi` and the narrowing induction `ni` discussed later, as, respectively, `gsi!` and `ni!`, which basically apply the `EPS` rule to each of the subgoals generated by the given original command. With the `gsi!` command, the proof in Section 2.2 is much simpler.[13]

```
NuITP> set module PEANO+R .

  Module PEANO+R is now active.

NuITP> genset GEN-NAT for Nat is 0 ;; s N:Nat .

  Generator set GEN-NAT for sort Nat added.

  GEN-NAT (default):
    0
    s N:Nat

NuITP> set goal X:Nat + (Y:Nat + Z:Nat) = (X:Nat + Y:Nat) + Z:Nat .

  Initial goal set.

  Goal Id: 0
  Generated By: init
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $1:Nat + ($2:Nat + $3:Nat) = ($1:Nat + $2:Nat) + $3:Nat

NuITP> apply gsi! to 0 on $3 with 0 ;; s(K:Nat) .

  Generator Set Induction (GSI) with Equality Predicate Simplification applied to goal 0.

  Goals 0.1.1 and 0.2.1 have been proved.

  qed
```

Notice that the message indicates that goals `0.1.1` and `0.2.1` have been proved. The application of the GSI rule generates goals `0.1` and `0.2`. Then, their simplification using EPS produces these other goals which get proven. We can see all the goals internally generated by using the `show goals` command.

[13] If the module to work with is already set as active module and the generator set to be used is already defined, it is not necessary to redefine it once and again, however, for these first examples, we will explicitly add `set module` and `genset` commands.

2.4 Proving commutativity of addition

In this section, we present a proof of the commutativity of addition as defined in the PEANO+R module. We can begin by setting the module as the active one, and then setting the goal to prove.

```
NuITP> set module PEANO+R .

  Module PEANO+R is now active.

NuITP> genset GEN-NAT for Nat is 0 ;; s N:Nat .

  Generator set GEN-NAT for sort Nat added.

  GEN-NAT (default):
    0
    s N:Nat

NuITP> set goal X:Nat + Y:Nat = Y:Nat + X:Nat .

  Initial goal set.

  Goal Id: 0
  Generated By: init
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $1:Nat + $2:Nat = $2:Nat + $1:Nat
```

Given this goal, we may begin by attempting to apply generator-set induction on one of the variables, say `$1:Nat`, using the generator set we have already used in previous proofs.

```
NuITP> apply gsi! to 0 on $1 with 0 ;; s K:Nat .

  Generator Set Induction (GSI) with Equality Predicate Simplification applied to goal 0.

  Goal Id: 0.1.1
  Generated By: EPS
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $2:Nat = 0 + $2:Nat

  Goal Id: 0.2.1
  Generated By: EPS
  Skolem Ops:
    $3.Nat
  Executable Hypotheses:
    $3 + $2:Nat => $2:Nat + $3
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    s ($2:Nat + $3) = s $3 + $2:Nat
```

In this case, the goals produced by the GSI rule are not proven after their simplification, and we are left with goals `0.1.1` and `0.2.1`. We can try to solve them by using induction again. Let us begin with Goal `0.1.1`.

```
NuITP> apply gsi! to 0.1.1 on $2 with 0 ;; s K:Nat .

  Generator Set Induction (GSI) with Equality Predicate Simplification applied to goal 0.1.1.

  Goals 0.1.1.1.1 and 0.1.1.2.1 have been proved.

  Unproven goals:

  Goal Id: 0.2.1
  Generated By: EPS
  Skolem Ops:
    $3.Nat
  Executable Hypotheses:
    $3 + $2:Nat => $2:Nat + $3
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    s ($2:Nat + $3) = s $3 + $2:Nat

  Total unproven goals: 1
```

Goal `0.1.1` has been automatically discharged, and we are left with Goal `0.2.1`.

```
NuITP> apply gsi! to 0.2.1 on $2 with 0 ;; s K:Nat .

  Generator Set Induction (GSI) with Equality Predicate Simplification applied to goal 0.2.1.

  Goal Id: 0.2.1.1.1
  Generated By: EPS
  Skolem Ops:
    $3.Nat
  Executable Hypotheses:
    $3 + $2:Nat => $2:Nat + $3
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $3 = 0 + $3

  Goal Id: 0.2.1.2.1
  Generated By: EPS
  Skolem Ops:
    $3.Nat
    $4.Nat
  Executable Hypotheses:
    $3 + $2:Nat => $2:Nat + $3
    s $3 + $4 => s ($4 + $3)
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    s ($4 + $3) = s $4 + $3
```

These goals highlight the fact that the equations defining the _+_ operator recurse on the right term, with no equations specifying what to do in these cases. Furthermore, no variables are left to apply induction on. We can, however, restart the goal and then apply induction. First on Goal `0.2.1.1.1`:

```
NuITP> apply rst to 0.2.1.1.1 .

  Restart (RST) applied to goal 0.2.1.1.1.

  Goal Id: 0.2.1.1.1.1
  Generated By: RST
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $3:Nat = 0 + $3:Nat

NuITP> apply gsi! to 0.2.1.1.1.1 .

  Generator Set Induction (GSI) with Equality Predicate Simplification applied to goal 0.2.1.1.1.1.

  Goals 0.2.1.1.1.1.1.1 and 0.2.1.1.1.1.2.1 have been proved.

  Unproven goals:

  Goal Id: 0.2.1.2.1
  Generated By: EPS
  Skolem Ops:
    $3.Nat
    $4.Nat
  Executable Hypotheses:
    $3 + $2:Nat => $2:Nat + $3
    s $3 + $4 => s ($4 + $3)
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    s ($4 + $3) = s $4 + $3

  Total unproven goals: 1
```

And then on Goal `0.2.1.2.1`:

```
NuITP> apply rst to 0.2.1.2.1 .

  Restart (RST) applied to goal 0.2.1.2.1.

  Goal Id: 0.2.1.2.1.1
  Generated By: RST
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    s ($4:Nat + $3:Nat) = s $4:Nat + $3:Nat

NuITP> apply gsi! to 0.2.1.2.1.1 .

  Generator Set Induction (GSI) with Equality Predicate Simplification applied to goal 0.2.1.2.1.1.

  Goals 0.2.1.2.1.1.1.1, 0.2.1.2.1.1.2.1, 0.2.1.2.1.1.3.1 and 0.2.1.2.1.1.4.1 have been proved.

  qed
```

In the coming sections we will see how, instead of 'restarting' the remaining goals, they are previously proved and then internalized, or introduced as lemmas 2.5. Again, we can ask the tool to print all internal goals with the `show goals` or `show goal` <*goal-id*> commands.

2.5 Commutativity of addition with lemmas

In this section, we illustrate the use of lemmas. Specifically, we prove the commutativity of addition in the `PEANO+R` module using lemmas `0 + N:Nat = N:Nat` and `s M:Nat + N:Nat = s(M:Nat + N:Nat)`. First, we set up the proof.

```
  NuITP> set module PEANO+R .

    Module PEANO+R is now active.

  NuITP> genset GEN-NAT for Nat is 0 ;; s N:Nat .

    Generator set GEN-NAT for sort Nat added.

    GEN-NAT (default):
      0
      s N:Nat

  NuITP> set goal X:Nat + Y:Nat = Y:Nat + X:Nat .

    Initial goal set.

    Goal Id: 0
    Generated By: init
    Skolem Ops:
      None
    Executable Hypotheses:
      None
    Quasi-Executable Hypotheses:
      None
    Non-Executable Hypotheses:
      None
    Goal:
      $1:Nat + $2:Nat = $2:Nat + $1:Nat
```

First, we enrich Goal 0 with the lemma `0 + N:Nat = N:Nat`. This produces two subgoals: Goal `0.1` is the introduced lemma and Goal `0.2` the original goal enriched with the lemma. The lemma can be proved with a single application of the `GSI` rule.

```
NuITP> apply le to 0 with 0 + N:Nat = N:Nat .

  Lemma Enrichment (LE) applied to goal 0.

  Goal Id: 0.1
  Generated By: LE
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $3:Nat = 0 + $3:Nat

  Goal Id: 0.2
  Generated By: LE
  Skolem Ops:
    None
  Executable Hypotheses:
    0 + $3:Nat => $3:Nat
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $1:Nat + $2:Nat = $2:Nat + $1:Nat

NuITP> apply gsi! to 0.1 on $3 .

  Generator Set Induction (GSI) with Equality Predicate Simplification applied to goal 0.1.

  Goals 0.1.1.1 and 0.1.2.1 have been proved.

  Unproven goals:

  Goal Id: 0.2
  Generated By: LE
  Skolem Ops:
    None
  Executable Hypotheses:
    0 + $3:Nat => $3:Nat
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $1:Nat + $2:Nat = $2:Nat + $1:Nat

  Total unproven goals: 1
```

Then, we enrich Goal `0.2` with the lemma `s M:Nat + N:Nat = s(M:Nat + N:Nat)`. Again, the lemma can be proved with a single application of the `GSI` rule.

```
NuITP> apply le to 0.2 with s M:Nat + N:Nat = s(M:Nat + N:Nat) .

  Lemma Enrichment (LE) applied to goal 0.2.

  Goal Id: 0.2.1
  Generated By: LE
  Skolem Ops:
    None
  Executable Hypotheses:
    0 + $3:Nat => $3:Nat
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    s ($4:Nat + $5:Nat) = s $4:Nat + $5:Nat

  Goal Id: 0.2.2
  Generated By: LE
  Skolem Ops:
    None
  Executable Hypotheses:
    0 + $3:Nat => $3:Nat
    s $4:Nat + $5:Nat => s ($4:Nat + $5:Nat)
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $1:Nat + $2:Nat = $2:Nat + $1:Nat

NuITP> apply gsi! to 0.2.1 on $5 .

  Generator Set Induction (GSI) with Equality Predicate Simplification applied to goal 0.2.1.

  Goals 0.2.1.1.1 and 0.2.1.2.1 have been proved.

  Unproven goals:

  Goal Id: 0.2.2
  Generated By: LE
  Skolem Ops:
    None
  Executable Hypotheses:
    0 + $3:Nat => $3:Nat
    s $4:Nat + $5:Nat => s ($4:Nat + $5:Nat)
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $1:Nat + $2:Nat = $2:Nat + $1:Nat

  Total unproven goals: 1
```

```
NuITP> apply gsi! to 0.2.2 on $1 .

  Generator Set Induction (GSI) with Equality Predicate Simplification applied to goal 0.2.2.

  Goals 0.2.2.1.1 and 0.2.2.2.1 have been proved.

  qed
```

2.6 Proving commutativity of addition in one shot

The **GSI** rule applied on multiple variables allows us to provide a simpler proof than that presented in Section 2.4 for the commutativity of addition. As always, we begin by setting the module as the active one, the generator set, and the goal. The RPO defined in the `PEANO+R` module has already been checked in previous examples.

```
NuITP> set module PEANO+R .

  Module PEANO+R is now active.

NuITP> genset GNAT for Nat is 0 ;; s K:Nat .

  Generator set GNAT for sort Nat added.

  GNAT (default):
    0
    s K:Nat

NuITP> set goal X:Nat + Y:Nat = Y:Nat + X:Nat .

  Initial goal set.

  Goal Id: 0
  Generated By: init
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $1:Nat + $2:Nat = $2:Nat + $1:Nat
```

We can then apply the **GSI** on multiple variables as follows.

```
NuITP> apply gsi! to 0 on $1:Nat $2:Nat .

  Generator Set Induction (GSI) with Equality Predicate Simplification applied to goal 0.

  Goals 0.1.1, 0.2.1 and 0.4.1 have been proved.

  qed
```

If we want **GSI** to be applied on all the variables in the goal, we can alternatively use the command without specifying the variables, as follows:

```
NuITP> undo 0 .

  Goal 0 undone. All its children have been deleted.

  Unproven goals:

  Goal Id: 0
  Generated By: init
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $1:Nat + $2:Nat = $2:Nat + $1:Nat

  Total unproven goals: 1

NuITP> apply gsi! to 0 .

  Generator Set Induction (GSI) with Equality Predicate Simplification applied to goal 0.

  Goals 0.1.1, 0.2.1 and 0.4.1 have been proved.

  qed
```

2.7 Proving commutativity and associativity of addition in one shot

The use of multiclauses allows us to proof together several goals. The possibility of applying the **GSI** rule on multiple variables allows us to proof associativiy and commutativity of addition in one single shot.[14]

```
NuITP> set module PEANO+R .

  Module PEANO+R is now active.

NuITP> genset GEN-NAT for Nat is 0 ;; s N:Nat .

  Generator set GEN-NAT for sort Nat added.

  GEN-NAT (default):
    0
    s N:Nat

NuITP> set goal X:Nat + Y:Nat = Y:Nat + X:Nat \
>            /\ X1:Nat + (Y1:Nat + Z:Nat) = (X1:Nat + Y1:Nat) + Z:Nat .

  Initial goal set.

  Goal Id: 0
  Generated By: init
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $2:Nat + $4:Nat = $4:Nat + $2:Nat /\ $1:Nat + ($3:Nat + $5:Nat) = ($1:Nat + $3:Nat) + $5:Nat

NuITP> apply gsi! to 0 .

  Generator Set Induction (GSI) with Equality Predicate Simplification applied to goal 0.

  Goals 0.1.1, 0.10.1, 0.11.1, 0.12.1, 0.13.1, 0.14.1, 0.15.1, 0.16.1, 0.17.1, 0.18.1,
    0.19.1, 0.2.1, 0.20.1, 0.21.1, 0.22.1, 0.23.1, 0.24.1, 0.27.1, 0.28.1, 0.29.1, 0.3.1,
    0.30.1, 0.31.1, 0.32.1, 0.5.1, 0.6.1, 0.7.1 and 0.8.1 have been proved.
```

[14]Please, note the use of the backslash to enter multi-line inputs (see Section 6.1).

```
  qed
```

2.8 Proving program equivalence

Using NuITP, we can prove that the PEANO+R module presented in Section 2.2 and the module PEANO+L below are *equivalent*, that is, that they compute the same addition function. We can do so by proving in PEANO+R the axioms in PEANO+L and vice versa.[15]

```
set include BOOL off .

fmod PEANO+L is
  sorts Nat NzNat .
  subsorts NzNat < Nat .

  op 0 : -> Nat [ctor rpo 1] .
  op s_ : Nat -> NzNat [ctor rpo 2] .

  op _+_ : Nat Nat -> Nat [rpo 3] .
  eq 0 + N:Nat = N:Nat .
  eq s N:Nat + M:Nat = s(N:Nat + M:Nat) .
endfm
```

Let us begin with the axioms of PEANO+L.

```
NuITP> set module PEANO+R .

  Module PEANO+R is now active.

NuITP> set goal 0 + Y:Nat = Y:Nat /\ s X:Nat + Y:Nat = s(X:Nat + Y:Nat) .

  Initial goal set.

  Goal Id: 0
  Generated By: init
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $2:Nat = 0 + $2:Nat /\ s ($1:Nat + $2:Nat) = s $1:Nat + $2:Nat

NuITP> apply gsi! to 0 on $2 with 0 ;; s K:Nat .

  Generator Set Induction (GSI) with Equality Predicate Simplification applied to goal 0.

  Goals 0.1.1 and 0.2.1 have been proved.

  qed
```

Then we can prove the axioms of PEANO+R in the module PEANO+L.

```
NuITP> set module PEANO+L .

  Module PEANO+L is now active.
```

[15]For a general notion of equivalence between equational programs and a justification of the proof method see [15].

```
NuITP> genset GEN-NAT for Nat is 0 ;; s N:Nat .

  Generator set GEN-NAT for sort Nat added.

  GEN-NAT (default):
    0
    s N:Nat

NuITP> set goal Y:Nat + 0 = Y:Nat /\ Y:Nat + s X:Nat = s(Y:Nat + X:Nat) .

  Initial goal set.

  Goal Id: 0
  Generated By: init
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $2:Nat = $2:Nat + 0 /\ s ($2:Nat + $1:Nat) = $2:Nat + s $1:Nat

NuITP> apply gsi! to 0 on $2 with 0 ;; s K:Nat .

  Generator Set Induction (GSI) with Equality Predicate Simplification applied to goal 0.

  Goals 0.1.1 and 0.2.1 have been proved.

  qed
```

With these two simple proof scripts we have shown the equivalence of `PEANO+R` and `PEANO-L`. Specifically, this proves that both modules have the same initial algebra and therefore satisfy the same inductive properties.

2.9 Enriched specifications

Since `PEANO+R` and `PEANO-L` are equivalent, we can define a new module `PEANO+LR` defined with the share signature of `PEANO+R` and `PEANO-L` and with all the equations of both modules without changing their initial semantics.

```
set include BOOL off .

fmod PEANO+LR is
  protecting PEANO+R .

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

Indeed, this is very similar to what we may get by lemma enrichment along our proofs, but it has the important advantage of "internalizing" such lemmas, so that they can be reused on many occasions. NuITP can help us with this internalization by using its internalization commands. Several commands will allow us to enrich the state of the proof, by either adding them as hypotheses for the rest of the session, or by modifying the active module by adding proven goals as either equations or axioms. That is, the tool will enrich the module without having to explicitly create a modified one ourselves. These options are presented and illustrated in the following sections. Further details can be found in Section 4.4.

2.9.1 Commutativity and associativity of addition on an enriched module

Once the proof of some goal has been completed, we can use the internalization commands to add it either as an axiom (using the `internalize as comm | assoc | assoc comm` command) or as equations or hypotheses (using the `internalize` command) depending on whether the axiom can be oriented or not. For example, right after the completion of the proof of the axioms of `PEANO-L` in the `PEANO-R` module shown in Section 2.8, we can internalize them to prove some new goals.

```
NuITP> set module PEANO+R .

  ...

NuITP> set goal 0 + Y:Nat = Y:Nat /\ s X:Nat + Y:Nat = s(X:Nat + Y:Nat) .

  ...

NuITP> apply gsi! to 0 on $2 with 0 ;; s K:Nat .

  ...

  qed
```

```
NuITP> internalize .

  Proof has been internalized.
```

Since the formulas in the goal can be oriented using the provided RPO, they have been added as equations to our module. The `PEANO+R` module has now been extended with equations

```
eq 0 + Y:Nat = Y:Nat .
eq s X:Nat + Y:Nat = s(X:Nat + Y:Nat) .
```

We can see the extended module with the `show internal module` command (see Section 4.1.2).

```
NuITP> show internal module .

fmod PEANO+R is
  pr NuITP-CLAUSE .
  sorts Nat NzNat .
  subsort NzNat < Nat .
  op 0 : -> Nat [ ctor rpo 1 ] .
  op _+_ : Nat Nat -> Nat [ rpo 3 ] .
  op s_ : Nat -> NzNat [ ctor rpo 2 ] .
  eq [I0] : 0 + H0:Nat = H0:Nat [ metadata "internal" ] .
  eq [I1] : s H1:Nat + H2:Nat = s (H1:Nat + H2:Nat) [ metadata "internal" ] .
  eq N:Nat + 0 = N:Nat .
  eq N:Nat + s M:Nat = s (N:Nat + M:Nat) .
endfm
```

We can see what and how has been internalized with the `show internal` command (see Section 4.1.2).

```
NuITP> show internal .

  Internalized equations:

  eq [I0] : 0 + H0:Nat = H0:Nat [ metadata "internal" ] .
  eq [I1] : s H1:Nat + H2:Nat = s (H1:Nat + H2:Nat) [ metadata "internal" ] .

  Internalized hypotheses:
```

```
  No hypotheses internalized.

  Locally internalized goals:

  No local goals internalized.
```

In this enriched module, we can now carry on proofs of associativity and commutativity of addition in one single step.[16]

```
NuITP> set goal (X:Nat + Y:Nat = Y:Nat + X:Nat) \
>             /\ ((X:Nat + Y:Nat) + Z:Nat = X:Nat + (Y:Nat + Z:Nat)) .

  Initial goal set.

  Goal Id: 0
  Generated By: init
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $1:Nat + $2:Nat = $2:Nat + $1:Nat /\ $1:Nat + ($2:Nat + $3:Nat) = ($1:Nat + $2:Nat) + $3:Nat

NuITP> apply gsi! to 0 on $2 with 0 ;; s K:Nat .

  Generator Set Induction (GSI) with Equality Predicate Simplification applied to goal 0.

  Goals 0.1.1 and 0.2.1 have been proved.

  qed
```

Since the goal proven specifies the commutativity and associativity of addition, once it has been proved, we can internalize it as `assoc` and `comm` attributes.

```
NuITP> internalize as assoc comm .

  Proven goal has been internalized as assoc and comm axioms.
```

2.9.2 Proving distributivity of multiplication over addition

Let us illustrate the use of axiom internalization in a proof of the distributivity of multiplication over addition. For that, let us consider the following `PEANO+x` module that defines operations for addition and multiplication.

```
set include BOOL off .

fmod PEANO+x is
  sorts Nat NzNat .
  subsorts NzNat < Nat .

  op 0 : -> Nat [ctor rpo 0] .
  op s_ : Nat -> NzNat [ctor rpo 1] .
```

[16]Note that the input has been broken into several lines to facilitate its readability. Any command input to NuITP must be entered in a single line (without carriage return). However, Maude supports backslash continuations as in shells, Python, etc. (see Section 6.1).

```
  vars N M : Nat .
  vars N' M' K' : NzNat .

  op _+_ : Nat Nat -> Nat [rpo 2] .
  eq N + 0 = N .
  eq N + s M = s(N + M) .

  op _*_ : Nat Nat -> Nat [prec 40 rpo 3] .
  eq N * 0 = 0 .
  eq N * s M = N + (N * M) .
endfm
```

You can try to prove distributivity directly. Instead, here we prove that addition is commutative and associative, internalize these formulas as axioms, and then will prove our intended goal.

First, as always, we set the PEANO+x module as the active module. To avoid explicitly giving the generator set for variables of sort Nat, we provide a new command genset to define a generator set labelled GEN-NAT that, since it is the first defined for sort Nat, will be used as the default generator set for that sort. That is, with a generator set defined, we can use the commands requiring one without explicitly giving it each time.

```
NuITP> set module PEANO+x .

  Module PEANO+x is now active.

NuITP> genset GEN-NAT for Nat is 0 ;; s N:Nat .

  Generator set GEN-NAT for sort Nat added.  GEN-NAT (default):
    0
    s N:Nat
```

Then, we prove commutativity and associativity of addition and internalize it as in Section 2.9.1.

```
NuITP> set goal (X:Nat + Y:Nat = Y:Nat + X:Nat) \
>             /\ ((X:Nat + Y:Nat) + Z:Nat = X:Nat + (Y:Nat + Z:Nat)) .

  Initial goal set.

  Goal Id: 0
  Generated By: init
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $1:Nat + $2:Nat = $2:Nat + $1:Nat /\ $1:Nat + ($2:Nat + $3:Nat) = ($1:Nat + $2:Nat) + $3:Nat

NuITP> apply gsi! to 0 .

  Generator Set Induction (GSI) with Equality Predicate Simplification applied to goal 0.

  Goals 0.1.1, 0.2.1, 0.3.1, 0.4.1, 0.5.1, 0.6.1, 0.7.1 and 0.8.1 have been proved.

  qed

NuITP> internalize as assoc comm .
```

```
  Proven goal has been internalized as comm and assoc axioms.
```

We are now ready to prove distributivity.

```
NuITP> set goal X:Nat * (Y:Nat + Z:Nat) = X:Nat * Y:Nat + X:Nat * Z:Nat .

  Initial goal set.

  Goal Id: 0
  Generated By: init
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $1:Nat * ($3:Nat + $2:Nat) = $1:Nat * $3:Nat + $1:Nat * $2:Nat
```

The tool can prove the goal by induction on variable $2.

```
NuITP> apply gsi! to 0 on $2:Nat .

  Generator Set Induction (GSI) with Equality Predicate Simplification applied to goal 0.

  Goals 0.1.1 and 0.2.1 have been proved.

  qed
```

2.9.3 Proving a program optimization

On the enriched module of Section 2.9.1, we show how to prove a different kind of goal, one that proves the correctness of a program optimization that will make the computation of addition faster. In this case, we just need equational simplification.[17]

```
NuITP> set module PEANO+R .

  ...

NuITP> set goal 0 + Y:Nat = Y:Nat /\ s X:Nat + Y:Nat = s(X:Nat + Y:Nat) .

  ...

NuITP> apply gsi! to 0 on $2:Nat with 0 ;; s K:Nat .

  ...

  qed

NuITP> internalize .

  Proof has been internalized.

NuITP> set goal (s X:Nat + s Y:Nat = s s X:Nat + Y:Nat) \
>            /\ (s s X:Nat + s s Y:Nat = s s s s(X:Nat + Y:Nat)) \
>            /\ (s s s X:Nat + s s s Y:Nat = s s s s s s X:Nat + Y:Nat) .
```

[17]We do not need to repeat the proof and the internalization; but to make sure that there is no confusion, we explicitly include the proof of the goal and its internalization before proceeding with the proof on the new goal.

```
  Initial goal set.

  Goal Id: 0
  Generated By: init
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    s s s s ($1:Nat + $2:Nat) = s s $1:Nat + s s $2:Nat /\ s $1:Nat + s $2:Nat = s s
    $1:Nat + $2:Nat /\ s s s $1:Nat + s s s $2:Nat = s s s s s s $1:Nat + $2:Nat

NuITP> apply eps to 0 .

  Equality Predicate Simplification (EPS) applied to goal 0.

  Goal 0.1 has been proved.

  qed
```

Once proven, the optimization can then be internalized to speed up later computations of addition.

3 Simplification strategies

NuITP provide commands for defining and using simplification strategies.

3.1 How to use user-defined strategies

The `!` commands presented in the previous sections illustrate the simplest form of strategy. Since the simplification using EPS of the generated goals by any command is quite useful, the `!` commands automate their simplification. For example, an `ni!` will apply the `NI` rule followed by the simplification of each of the goals generated by it using the EPS rule. There are several examples using these `!` commands in the previous sections, see, e.g., those in Sections 2.3 or 2.4. As explained in Section 3.2, more complex strategies can also be defined. These user-defined strategies can then be used in three different ways.

1. The `apply` commands provide a variant in which a strategy identifier can be specified, so that it will be applied after the corresponding inference rule. E.g., the command

   ```
   NuITP> apply gsi{MY-STRAT} to 0.1 on $1 with 0 ;; s Y:Nat .
   ```

   would apply the **GSI** rule followed by the application of the simplification strategy `MY-STRAT` on each generated subgoal. See Section 4.3.1 for details on how to define a new strategy.

2. The `*` commands do the same, but applying the default strategy.

   ```
   NuITP> apply gsi* to 0.1 on $1 with 0 ;; s Y:Nat .
   ```

   See Section 4.3.2 for details on how to declare a user-defined strategy the default one.

3. The `simplify` command applies a user-defined strategy on the specified goal. See Section 4.3.3.

3.2 Defining more complex strategies

As of NuITP alpha 36, try-versions of each simplification rule are provided so the user can combine then, by concatenating them with the semicolon char, to create new, more complex, simplification strategies. The following are the most basic units that can be used to create new strategies:

`try-eps`	`try-subl`	`try-cs`	`try-rst`
`try-cvul`	`try-subr`	`try-icc`	
`try-cvufr`	`try-ns`	`try-gnd`	

Even though `GSI` and `CAS` are not a simplification rules, if the sorts of all the variables in the corresponding goal have a default generator set defined, the `try-gsi` and `try-cas` tactics, applying induction or cases, respectively, on all variables are also available.

Each of these basic tactics attempts the corresponding rule, and if succeeds, then applies a `try-eps` — except for the `try-eps` itself, of course. If the application of the rule fails, then the goal is left unchanged.

NuITP provides a default simplification strategy with identifier `NuITP`, as shown in the following example:

```
     ===================================
                   NuITP
          Inductive Theorem Prover
       for Maude Equational Theories
        (alpha 36 built Mar 8th 2026)
     ===================================
             Copyright 2021-2026
     Universitat Politècnica de València
     ===================================

NuITP> show strats .

  Simplification strategies:

  NuITP (default): try-cs ; try-cvul ; try-cs ; try-icc ; try-cs.

NuITP> strat MY-STRAT is try-cs ; try-cvul ; try-icc ; try-gnd ; try-icc ; try-cs .

  Simplificaiton strategy MY-STRAT added successfully.

NuITP> show strats .

  Simplification strategies:

  NuITP (default): try-cs ; try-cvul ; try-cs ; try-icc ; try-cs.

  MY-STRAT: try-cs ; try-cvul ; try-icc ; try-gnd ; try-icc ; try-cs.

NuITP> set default strat MY-STRAT .

  Simplificaiton strategy MY-STRAT is now default.

NuITP> show strats .

  Simplification strategies:

  MY-STRAT (default): try-cs ; try-cvul ; try-icc ; try-gnd ; try-icc ; try-cs.

  NuITP: try-cs ; try-cvul ; try-cs ; try-icc ; try-cs.
```

Note that NuITP requires to apply the simplification rules inside a strategy by using the above try-versions, as they will automatically perform the necessary steps to unfold the goal while taking into account how the new frontier of the current proof evolves via concurrent rewriting. Also note that the `try-gsi` and `try-cas` strategies are actually based on induction rules, namely Generator Set Induction (`GSI`) and Case (`CAS`), which usually take some arguments to be able to apply them. In both cases, the strategies will be able to apply the corresponding induction rule if they meet the requirements for an automated application, i.e., a default generator set is defined for each of its sorts.

Eventually, NuITP will provide more powerful facilities for the definition of new strategies in a richer strategy language, which may even provide access to Maude's strategy language [3]. However, even though this general mechanism is not available yet, as the `!` commands do for `EPS`, the `*` commands invoke a predefined strategy on each of the goals generated by the corresponding command. Moreover, as users may now define their own strategies with the above basic atoms, they can apply them by either setting them to the default strategy to apply when running a `*` command or by explicitly enclosing them in brackets after the rule name without blank spaces (e.g., `apply ns{MY-STRAT} to ...`).

4 NuITP commands

In this section, we describe all the commands, including their syntax, an example of use, and some requirements that must be satisfied. Specifically, in Section 4.1 we present NuITP's general commands. In Section 4.2, we explain the generator-set related commands. Section 4.4 explains the different internalization commands. In Section 4.5 we introduce the simplification commands, which require less user interaction. Finally, in Section 4.6 we introduce the induction commands, which may require more user interaction and add induction hypotheses to the current goal.

4.1 General NuITP commands

4.1.1 Initialization commands

The first task when starting a new NuITP session is to provide both the theory in which we want to prove a goal and the goal itself, which we can achieve by means of the `set` command.

First, we set the module that NuITP will use in the current session as follows:

`set module` <*module-name*> `.`

where <*module-name*> is the identifier of the Maude module we want to set as current module in NuITP. Note that the module must have been previously loaded into the Maude interpreter and be available in the Maude System database.

Next, we set an initial goal to be proven with the following command:

`set goal` <*goal*> `.`

where <*goal*> is a multiclause of the form Γ `->` Λ, where Γ is a conjunction of equations and Λ a conjunction of disjunctions of equations see Section 1.3.

Consider the already-discussed `PEANO+R` module, which has been previously loaded in the Maude system. First, we set `PEANO+R` as the theory that NuITP will use in this session:

```
NuITP> set module PEANO+R .

   Module PEANO+R is now active.
```

Once we have loaded the theory, we set an initial goal to be proved. For example, if we want to prove a simple, unconditional goal, we can write the following:

```
NuITP> set goal s X:Nat + Y:Nat = s(X:Nat + Y:Nat) .

  Initial goal set.

  Goal Id: 0
  Generated By: init
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    s ($1:Nat + $2:Nat) = s $1:Nat + $2:Nat
```

Other goals may be more complex: they may be conditional and may have a conclusion which is a conjunction of disjunctions of equations.

4.1.2 Information commands

The show commands show different kinds of information related to the state of the prover.

- The `show log` command shows a raw log of the session.
- The `show session log` command shows the entire NuITP session log.
- The `show module` command shows the currently active module.
- The `show local module` command shows the active module including locally internalized executable lemmas.
- The `show extended module` command shows the active module extended with equality predicates.
- The `show internal module` command shows the active module including all internalized executable lemmas.
- The `show goal` <*goal-id*> shows the goal with the specified identifier.
- The `show goals` command shows all goals, that is, the entire proof tree.
- The `show frontier` shows the goals in the frontier, that is, the goals pending to be proved to complete the proof of the current top goal.
- The `show gensets` command shows all the generator sets defined by the user in the current session (see Section 4.2).
- The `show gensets for` <*sort*> command shows the defined generator sets for the specified sort.
- The `show genset` <*gen-set-id*> command shows the specified generator set.
- The `set verbose (on | off)` command enables or disables the verbosity of some of the `show` commands above. By default, the verbose option is enabled.
- The `show strats` command shows all the defined simplification strategies in the current session (see Section 3).

Syntax of the commands

```
show [session] log .
show [local | extended | internal] module .
show goal <goal-id> .
show goals .
show frontier .
show gensets [for <sort-name>] .
show genset <gen-set-id> .
set verbose (on | off) .
```

Examples of the use of each command

```
NuITP> show module .
```

```
NuITP> show log .
NuITP> show goals .
```

```
NuITP> show frontier .
```

```
NuITP> show goal 0.1 .
```

```
NuITP> show gensets .
```

```
NuITP> show gensets for Nat .
```

```
NuITP> show genset GEN-NAT .
```

```
NuITP> set verbose on .
```

```
NuITP> set verbose off .
```

4.1.3 Load and save proofs, snapshots, and reports

The `load`, `save`, `snap`, and `tex` commands allow loading and saving proof scripts and NuITP snapshots of the current session, as well as generating proof reports in LaTeX. Note that these commands do not have a final dot. For these commands to work, remember that you must execute Maude with the `allow-files` or `trust` flags (see Section 2.1).

The `load` command either loads (and runs) a proof script, which helps to automatize proofs, or restores a saved snapshot of a NuITP session, replacing the current one. Note that NuITP will automatically determine which functionality will be executed depending on the contents of the loaded file and, specifically, the first line (which must not be manually deleted by the user).

The counterpart of `load` are the `save`, `snap`, `tex`, and `html` commands, which respectively either (i) save a minimal[18] proof script of the current proof in the specified file, (ii) save a snapshot of the current NuITP session, which can be resumed later on by loading the file; or (iii) create a LaTeX / HTML5 report of the current proof and save it in the specified file.[19]

Note that, while loading NuITP scripts, blank lines will be ignored, as well as those starting with the `#` char, which can be used to write comments. Moreover, a single line consisting of the `eof` keyword will finish the script as if it reached the end of file, thus ignoring the lines below it.

Beware that NuITP does *not* ask for confirmation of the provided filename when saving either the session script or the LaTeX / HTML report into the provided file. It is thus the user responsibility to provide a safe filename.

Syntax of the commands

`load` <*file-name*>
`save` <*file-name*>
`snap` <*file-name*>
`tex` <*file-name*>
`html` <*file-name*>

Examples of the use of each command

[18] Where the effects of possible `undo` commands (see Section 4.1.4) have been applied.

[19] Note that NuITP will automatically add a `.proof`, `.snap`, `.tex`, or `.html` extension, respectively, to the provided file name if it is omitted.

```
NuITP> load scripts/my-script
```

```
NuITP> load sessions/my-session
```

```
NuITP> save scripts/my-script
```

```
NuITP> snap sessions/my-session
```

```
NuITP> tex reports/my-report
```

```
NuITP> html reports/my-report
```

Requirements of the commands

Maude has to be started with the `-allow-files` or `-trust` flags.

4.1.4 Undo and clear

The `undo` command, as its name indicates, undoes the effect of any simplification or inference rule applied to the goal named by the given identifier, automatically removing from the proof tree any goals derived from it or from any of its immediate children. For example, undoing goal `0` will reset the proof entirely up to the point in which we set the initial goal. Note that the goal named by the given identifier will not be removed. Instead, it becomes part of the current new frontier; and it will therefore be ready to have different rules applied to it.

Moreover, the `clear` and `clear all` commands reset the current goal or session, respectively. In particular, `clear` is equivalent to apply the `undo` command to goal with identifier 0, but also *forgets* the lemmas that may have been locally internalized while working in the current proof tree. In contrast, `clear all` resets the entire session, including internalization (local and global), generator sets, strategies, selected module, etc.

Syntax of the command

`undo` <*goal-id*> `.`
`clear` [`all`] `.`

where <*goal-id*> is the goal on which to apply the command.

Examples of the use of each command

```
NuITP> undo 0.1 .
```

```
NuITP> clear .
```

```
NuITP> clear all .
```

The `undo` command is used, e.g., in Section 2.6.

Requirements of the command

By definition, non-undoable goals, that is, goals where `undo` can *not* be applied, are “childless” goals, that is, either “closed” goals whose formula is `true`, or goals in the current frontier to which no inference rule has yet been successfully applied and therefore there is nothing to undo.

4.1.5 Assert

The `assert` commands help users to add checkpoints on their scripts in order to ensure that some specific goals or the entire current proofs have been successfully proved before continuing. Specifically, a user may want to check if a goal has been successfully proved without internalizing it (which would have the side-effect of checking if it is indeed fully proved up to the *qed* message) before moving on to the next one. Note that executing the `assert` command is actually equivalent to executing `assert 0`.

Syntax of the command

```
assert .
assert <goal-id> .
```

where <*goal-id*> is the goal on which to apply the command.

Examples of use of the command

```
NuITP> assert .
```

```
NuITP> assert 0.1 .
```

4.1.6 Help

The `help` command shows a brief summary of the available commands. Note that it does not expect a final dot.

```
NuITP> help
```

4.1.7 Quit

The `quit` command, abbreviated `q`, is used to leave the prover. It does not expect a final dot.

```
NuITP> quit
```

or

```
NuITP> q
```

4.2 Generator-set commands

As discussed in Section 1.7, one may use several generator sets for the same sort s, depending on the various functions that one may wish to reason inductively about. This section shows how to define new generator sets and how to set one of them as the default one for a given sort s, so that we do not have to explicitly provide a generator set or indicate the one we want to use for each command. Of course, if when giving a specific command we want to use a generator set different from the default one, we just need to mention its name when giving the command.

4.2.1 Adding a generator set

Syntax of the command

genset <*gen-set-id*> for <*sort*> is <*gen-set*> .

where <*gen-set-id*> is the identifier of the new generator set, <*sort*> is the sort associated with the generator set, and <*gen-set*> is a set of terms separated by double semicolons we want to use as a generator set for all ground constructor terms of the given sort.

Examples of use of the command

```
NuITP> genset GEN-NAT for Nat is 0 ;; s(N:Nat) .
```

Requirements of the command

The tool does not allow having generator sets with the same name for different sorts. However, we may redefine a generator set just by entering a new one using the same name. That is, if we have already defined a generator set in the same session with the same identifier, two things may happen, depending on whether the set is for the same sort or not. If the two generator sets are for the same sort, it will be updated with the new generator set. If we try to define a new generator set with the same identifier of an existing one but for a different sort, then an error message will be given.

The first generator set defined for a sort will be set as the default generator set for such a sort. If we wish to make a different generator set the default one for a given sort s, we can use the command explained in the next section.

4.2.2 Setting a generator set as default

Syntax of the command

set default genset <*gen-set-id*> .

where <*gen-set-id*> is the identifier of the new generator set we want to set as the default generator set for its corresponding sort.

Examples of use of the command

```
NuITP> genset GEN-NAT is default .
```

Requirements of the command

The generator set with the given identifier must have been previously defined.

4.3 Simplification strategy commands

This section presents the commands for adding a simplification strategy, setting a simplification strategy as default, and applying a simplification strategy.

4.3.1 Adding a simplification strategy

Syntax of the command

strat <*strat-id*> is <*strategy*> .

where <*strat-id*> is the identifier of the new strategy and <*gen-set*> is a NuITP simplification strategy, i.e., a sequence of NuITP simplification rules (or strategies) separated by semicolons.

Example of use of the command

```
NuITP> strat MY-STRAT is try-cs ; try-cvul ; try-cs ; try-icc ; try-cs .
```

Requirements of the command

The specified strategy must consist of a list of NuITP simplification strategies separated by semicolons. NuITP provides basic try-versions of each simplification rule, plus a default simplification strategy, which can be combined and extended with user-defined strategies to create new strategies. See more about NuITP strategies in Section 3.

4.3.2 Setting a simplification strategy as default

Syntax of the command

`set default strat` <*strat-id*> `.`

where <*strat-id*> is the identifier of the simplification strategy we want to set as the default simplification strategy.

Example of use of the command

```
NuITP> set default strat MY-STRAT .
```

Requirements of the command

The simplification strategy with the given identifier must have been previously defined.

4.3.3 Applying a simplification strategy

Any previously defined strategy can be applied to a goal using the `simplify` command.

Syntax of the command

`simplify` <*goal-id*> [`using` <*strat-id*>] `.`

where <*goal-id*> is the identifier of the goal we want to simplify and <*strat-id*> is the identifier of the simplification strategy we want to apply. When no strategy identifier is given, the current default simplification strategy will be applied.

Examples of use of the command

```
NuITP> simplify 0 .
```

```
NuITP> simplify 0 using MY-STRAT .
```

Requirements of the command

If explicitly given, the simplification strategy with the corresponding identifier must have been previously defined; otherwise, a default strategy must already exist.

4.4 Internalization commands

To prove and then use a previously proven result, we can either explicitly introduce it as a lemma in the proofs of the current goal, or we can use NuITP's *internalization* mechanism. The effect of the internalization mechanism is very similar to that of introducing a lemma: an already proven goal can be *internalized*, i.e., it can be automatically added to the current goal, exactly as a proved lemma would be, but without having to give and then discharge a lemma enrichment command. Furthermore, if the internalized result (i) contains a conjunction of (possibly conditional) equations, then, those *orientable* as rewrite rules in the specified RPO order are added as executable equations to the current module, whereas (ii) all other clauses or non-orientable conditional equations are added as non-executable hypothesis. Finally, (iii) if the result to be internalized is a fully general (i.e., not an instance of a more general one) axiom(s) of either associativity, commutativity, or associativity-commutativity for a binary operator f in the current goal not previously specified as such, then it can be given the attribute `assoc`, `comm`, or `assoc comm`, provided the existing RPO-orientable equations and hypotheses in the module remain RPO-terminating when such new axioms are added. The main difference between explicitly proved lemmas and internalized results is that the internalization mechanism persists along all proofs in the current NuITP session, while a lemma is only available in the current proof.

4.4.1 Internalization of a proven formula as equations and/or or hypotheses

Syntax of the command

`internalize` $\left[<\textit{goal-id}>\right]$ `.`

Examples of use of the command

```
NuITP> internalize .
```

```
NuITP> internalize 0.1 .
```

See examples of use in Sections 2.9.1 and 2.9.3.

Requirements of the command

By default, the initial goal is considered, unless explicitly providing the identifier of a subgoal of the current proof. The goal to be internalized has no Skolem constants and its proof has been successfully completed. If the current active goal to be internalized has the form:

$$(w_1 = w_1' \wedge \ldots \wedge w_n = w_n') \rightarrow ((u_1^1 = v_1^1 \vee \ldots \vee u_{m_1}^1 = v_{m_1}^1) \wedge \ldots \wedge (u_1^k = v_1^k \vee \ldots \vee u_{m_k}^k = v_{m_k}^k))$$

then the clauses in the goal will be internalized as different formulas:

$$(w_1 = w_1' \wedge \ldots \wedge w_n = w_n') \rightarrow (u_1^1 = v_1^1 \vee \ldots \vee u_{m_1}^1 = v_{m_1}^1)$$

$$\ldots$$

$$(w_1 = w_1' \wedge \ldots \wedge w_n = w_n') \rightarrow (u_1^k = v_1^k \vee \ldots \vee u_{m_k}^k = v_{m_k}^k).$$

Furthermore, if any of the above clauses is a (possibly conditional) equation that can be oriented as a rewrite rule using the specified RPO order, then it will be internalized as an (executable) equation. Otherwise, it will be added as a hypothesis, available for subsequent proofs.

4.4.2 Internalization of all intermediate proven goals

Syntax of the command

```
internalize all .
```

Examples of use of the command

```
NuITP> internalize all .
```

Requirements of the command

The command is given when the proof of the current active goal that we wish to internalize to use it as a lemma in the proof of subsequent goals has been successfully completed. Then, the already proved current goal itself, as well as each of its (obviously already proven) intermediate subgoals which do not have any Skolen constant are added to the current module exactly as done in the `internalize` command, thus becoming available as lemmas in the proofs or subsequent goals in the current NuITP session.

4.4.3 Internalization of proven formulas as axioms

If the goal proven corresponds to an associativity ($f(X,\ f(Y,\ Z)) = f(f(X,\ Y),\ Z)$), or commutativity ($f(X,\ Y) = f(Y,\ X)$) axiom, or is an associativity-commutativity conjunction $f(X,\ f(Y,\ Z)) = f(f(X,\ Y),\ Z) \wedge f(X,\ Y) = f(Y,\ X)$, and the axiom or axioms are fully general as explained above, not previously declared as axiom(s) for f, then the proved goal can be internalized as equational attribute(s) of f, provided the RPO-terminating and possibly conditional equations and hypotheses in the current module remain so after adding the new axiom(s).

Syntax of the command

```
internalize as (assoc | comm | assoc comm) .
```

Examples of use of the command

```
NuITP> internalize as assoc .
```

```
NuITP> internalize as comm .
```

```
NuITP> internalize as assoc comm .
```

Requirements of the command

The proof of the current active goal has been successfully completed. Furthermore, for the `internalize as comm` command, the goal to internalize must be of the form $f(X,\ Y) = f(Y,\ X)$. Similarly, for the `internalize as assoc` command, the goal to internalize must be of the form $f(X,\ f(Y,\ Z)) = f(f(X,\ Y),\ Z)$ or $f(f(X,\ Y),\ Z) = f(X,\ f(Y,\ Z))$. And finally, for the `internalize as assoc comm` command, the goal to internalize must be the conjunction of the above corresponding goals. In all cases, f must be a binary operator, all variables used in the goal (X, Y, Z) must be declared of the same sort s such that $f : s\ s \rightarrow s$ is the *most general* declaration for f in the current module, i.e., any other declaration $f : s'\ s'' \rightarrow s'''$ is such that $s', s'', s''' \leqslant s$.

4.4.4 Local internalization of intermediate goals

NuITP allows the automated local internalization of proven, Skolem-free subgoals generated in the current proof tree, i.e., those goals whose proof subtree is fully unfolded and all its leaves are true.

Syntax of the command

```
set local internalize (on | off) .
```

Examples of use of the command

```
NuITP> set local internalize on .
```

```
NuITP> set local internalize off .
```

Requirements of the command

The set of locally internalized subgoals will only be available for the current proof tree and setting a new initial goal will clear them off.

4.5 Simplification commands

In NuITP, simplification inference rules transform goals into simpler goals, sometimes proving them altogether. Since applying them is very often advantageous, they are ideally suited to be automated by means of strategies (see Sections 4.3 and 3). Furthermore, they require little to no user interaction. Most of them can be applied manually, and for most of them only the goal identifier in which we want to apply a given simplification rule is required.

4.5.1 Equality Predicate Simplification (EPS)

The initial algebra $T_{\mathcal{E}}$ of an equational program $\mathcal{E}$ satisfies various formula equivalences. For example, for $\mathcal{E}$ defining natural number addition the following equivalences hold: $0 = s(x) \equiv \bot$, $x = x \equiv \top$, $s(x) = s(y) \equiv x = y$, and $x+y = x+z \equiv y = z$. NuITP uses the theory transformation $\mathcal{E} \mapsto \mathcal{E}^{=}$ from [8] to automatically generate many of these equivalences as convergent rewrite rules to simplify goals. EPS applies the rules in $\mathcal{E}^{=}$ (which contains those in $\mathcal{E}$) and the goal's executable induction hypotheses, i.e., those RPO-orientable and those usable by *ordered rewriting* (see [16, §3.2]), to simplify a goal $\Gamma \to \Lambda$ to the goals in $(\Gamma \to \Lambda)!_{\vec{\mathcal{E}}^{=}_{\overline{X}_U} \cup \vec{H}^{+}_{e_U}}$, that is, the goals obtained by fully simplifying $\Gamma \to \Lambda$ with the rules in $\mathcal{E}^{=}$ and H. The hypotheses H have always been simplified beforehand (see [16, §3.2]).

$$\frac{\{[\overline{X},\mathcal{E},H] \Vdash \Gamma'_i \to \Lambda'_i\}_{i\in I}}{[\overline{X},\mathcal{E},H] \Vdash \Gamma \to \Lambda}$$

where $(\bigwedge_{i\in I} \Gamma'_i \to \Lambda'_i) \in (\Gamma \to \Lambda)!_{\vec{\mathcal{E}}^{=}_{\overline{X}_U} \cup \vec{H}^{+}_{e_U}}$.

If $\top \in (\Gamma \to \Lambda)!_{\vec{\mathcal{E}}^{=}_{\overline{X}_U} \cup \vec{H}^{+}_{e_U}}$, then $\bigwedge_{i\in I} \Gamma'_i \to \Lambda'_i$ is chosen to be $\top$. If $\bot \in (\Gamma \to \Lambda)!_{\vec{\mathcal{E}}^{=}_{\overline{X}_U} \cup \vec{H}^{+}_{e_U}}$, then $\bigwedge_{i\in I} \Gamma'_i \to \Lambda'_i$ is chosen to be $\bot$, i.e., the conjecture $\Gamma \to \Lambda$ is then shown to be *false*.

Example 1 (Borrowed from [16]) *Let $\mathcal{NP}$ denote theory of the natural numbers in Peano notation with the usual equations defining addition ($+$), multiplication ($*$), and exponentiation ($(\_)^{(\_)}$), and with two additional sorts, Pair and UPair, of, respectively, ordered and unordered pairs of numbers, built with the constructor operators, $[\_,\_] : Nat\ Nat \to Pair$ and $\{\_,\_\} : Nat\ Nat \to UPair$, where $\{\_,\_\}$ satisfies the* commutativity *axiom. Now consider the goal:*

$$[\varnothing,\mathcal{NP},\varnothing] \Vdash \{x^{s(s(0))},y\} = \{s(y),0\} \to [x + x^{s(s(0))}, x*x] = [s(x+y),x]$$

Note that the theory $\vec{\mathcal{NP}}^{=}$ includes, among others, the rules:

- $[x,y] = [x',y'] \to x = x' \wedge y = y'$
- $\{x,y\} = \{x',y'\} \to (x = x' \wedge y = y') \vee (x = y' \wedge y = x')$
- $s(x) = x \to \bot$.

Application of the first and second $\vec{\mathcal{NP}}^{=}$-rules to the above clause, plus Boolean simplification, plus the equations for $+$, $*$ and $(\_)^{(\_)}$, gives us a conjunction of two multiclauses:

$$x*x = s(y), y = 0 \to x + (x*x) = s(x+y) \wedge x*x = x$$

and

$$x*x = 0, y = s(y) \to x + (x*x) = s(x+y) \wedge x*x = x.$$

*Since $y = s(y)$ simplifies to $\bot$ by the third rule, the second multiclause simplifies to $\top$. So we just get the first multiclause as the result of applying the **EPS** rule to our original goal.*

See Section 2.9.3 for a proof of the above goal using NuITP.

The reader is referred to [16, §4.1] for additional details and examples on the **EPS** rule.

Syntax of the command

apply eps[! | * | {< *strat-id* >}] to < *goal-id* > .

where < *strat-id* > is the identifier of the strategy to be applied on the generated subgoals, and < *goal-id* > is the identifier of the goal that we want to simplify with the **EPS** rule.

Examples of use of the command

```
NuITP> apply eps to 0.1 .
```

```
NuITP> apply eps! to 0.1 .
```

```
NuITP> apply eps* to 0.1 .
```

```
NuITP> apply eps{MY-STRAT} to 0.1 .
```

Note that, applying rule **EPS** followed by **!** or a strategy consisting of the single application of (again) **EPS**, although possible, is generally useless, as the first application of the rule will compute the canonical form modulo the equality predicates theory. Using `eps*` or `eps{<strat-id>}` may be useful in some cases.

See examples of use of this inference rule in Sections 2.2, 2.9.3, 5.1, 5.2, and 5.3.

4.5.2 Constructor Variant Unification Left (CVUL)

The **CVUL** rule unifies those equations in the condition that belong to the FVP subtheory $\mathcal{E}_1$ (see Section 1.3). Then, each of the unifiers is applied to the rest of the multiclause creating a new subgoal. The most powerful case is when there is no solution for the unification problem, since then there are no subgoals, i.e., the rule's premise becomes $\top$, and we have *proved* the given goal. A more general version of the **CVUL** rule is presented in [16, §4.1]. Here we give a simpler, less general version that conveys the main ideas about this rule.

$$\frac{\{[\overline{X},\mathcal{E},H] \Vdash (\Gamma' \to \Lambda)\alpha\}_{\alpha \in \mathit{Unif}^{\Omega}_{\mathcal{E}_1}(\Gamma)}}{[\overline{X},\mathcal{E},H] \Vdash \Gamma,\Gamma' \to \Lambda}$$

where Γ is a conjunction of $\mathcal{E}_1$-equalities, Γ' does not contain any $\mathcal{E}_1$-equalities, and $\mathit{Unif}^{\Omega}_{\mathcal{E}_1}(\Gamma)$ denotes the set of constructor $\mathcal{E}_1$-unifiers of Γ [14, 18].

The generalized version of this rule presented in [16, §4.1], which is used by NuITP, makes this simplification rule *much more generally applicable*, since it handles the possibility of having in Γ some extra constants in $\overline{X}$. Let us consider the following example to better understand the rule.

Example 2 *Consider an equational theory $\mathcal{E}$ containing a specification of natural numbers, with sorts MBool, NzNat, and Nat, and a subsort inclusion NzNat < Nat. The sort MBool is defined with constructors True and False. And we have constructors* 0 *of sort Nat and* 1 *of sort NzNat, and overloaded operators* $\_+\_ : \mathit{NzNat}\ \mathit{NzNat} \to \mathit{NzNat}$*, which is a constructor for sort NzNat, and* $\_+\_ : \mathit{Nat}\ \mathit{Nat} \to \mathit{Nat}$*. Both* $\_+\_$ *operators are declared associative, commutative, and with identity element 0. Then, we have operators* $\_>\_ : \mathit{Nat}\ \mathit{Nat} \to \mathit{MBool}$ *and monus operator* $\_-\_ : \mathit{Nat}\ \mathit{Nat} \to \mathit{Nat}$*, defined by the variant equations* $N+M+1>N = \mathit{True}$*,* $N>N+M = \mathit{False}$*,* $(M+N)-N = M$*, and* $M-(N+M) = 0$*, with N and M variables of sort Nat. Now, given variables N and M also of sort Nat, consider the following goal:*

$$(X>Y = \mathit{True}) \to (X-Y>0 = \mathit{True}).$$

The unification of $(X > Y = True)$ *with the variant equations in the specification will lead to a unique solution* $\{X \mapsto N+M+1, Y \mapsto N\}$, *leaving us with a subgoal*

$$(N+M+1 > N = True) \to (N+M+1-N > 0 = True),$$

which can be discarded just by simplification using the ***EPS*** *rule. Please, see Section 5.7 for the Maude code and its proof script.*

Syntax of the command

```
apply cvul[! | * | {<strat-id>}] to <goal-id> .
```

where *<strat-id>* is the identifier of the strategy to be applied on the generated subgoals, and *<goal-id>* is the identifier of the goal that we want to simplify with the **CVUL** rule.

Recall that, here and in what follows, adding `!` (resp. `*`, resp. `{`*strat-id*`}`) after a rule name means applying `EPS` (resp. the default strategy, resp. the *strat-id* strategy) to each of the goals generated by the given command.

Examples of use of the command

```
NuITP> apply cvul to 0.1 .
```

```
NuITP> apply cvul! to 0.1 .
```

```
NuITP> apply cvul* to 0.1 .
```

```
NuITP> apply cvul{MY-STRAT} to 0.1 .
```

Requirements of the command

The condition of the multiclause contains at least one equality $u = v$ that is a Σ_1-formula, i.e., both u and v are Σ_1-terms (see again Section 1.3).

4.5.3 Constructor Variant Unification Failure Right (CVUFR)

The **CVUFR** rule may be seen as a restricted version of the more general **CVUL** rule.

$$\frac{[\overline{X}, \mathcal{E}, H] \Vdash \Gamma \to \Lambda \wedge \Delta}{[\overline{X}, \mathcal{E}, H] \Vdash \Gamma \to \Lambda \wedge (u = v, \Delta)}$$

where $u = v$ is a $\mathcal{E}_{1_{\overline{X}}}$-equality and $Unif^{\Omega}_{\mathcal{E}_1}((u = v)^{\circ}) = \varnothing$.

Syntax of the command

```
apply cvufr[! | * | {<strat-id>}] to <goal-id> .
```

where *<strat-id>* is the identifier of the strategy to be applied on the generated subgoals, and *<goal-id>* is the identifier of the goal that we want to simplify with the **CVUFR** rule.

Examples of use of the command

```
NuITP> apply cvufr to 0.1 .
```

```
NuITP> apply cvufr! to 0.1 .
NuITP> apply cvufr* to 0.1 .
```

```
NuITP> apply cvufr{MY-STRAT} to 0.1 .
```

Requirements of the command

The right-hand side of the multiclause contains at least one equality $\bar{u} = \bar{v}$ such that: (i) both u and v are Σ_1-terms; (ii) they may contain *Skolem* constants (denoted by writing $\bar{a}$ instead of a); and (iii) the set of constructor variant unifiers $\mathit{unif}^{\Omega}_{\mathcal{E}_1}(u = v)$ is empty (see Section 1.3).

4.5.4 Substitution Left (SUBL)

The substitution left (**SUBL**) rule basically substitutes equations of the form $x = u$ from the left-hand side of the multiclause, with x a variable, or $\overline{x} = u$, with $\overline{x}$ a (universal) Skolem constant.

When x is a variable, the inference rule is as follows.

$$\frac{[\overline{X}, \mathcal{E}, H] \Vdash (\Gamma \to \Lambda)\{x \mapsto u\}}{[\overline{X}, \mathcal{E}, H] \Vdash x = u, \Gamma \to \Lambda}$$

where: (i) x is a variable of sort s, $ls(u) \leqslant s$, and $x \notin \mathit{vars}(u)$; and (ii) u is not a Σ_1-term and Γ contains no other Σ_1-equations. Note that $=$ is assumed commutative, so cases $x = u$ and $u = x$ are both covered.

See [16] for the case in which $\overline{x}$ is a fresh constant.

Syntax of the command

apply subl$\big[$! | * | {<*strat-id*>}$\big]$ to <*goal-id*> .

where <*strat-id*> is the identifier of the strategy to be applied on the generated subgoals, and <*goal-id*> is the identifier of the goal that we want to simplify with the **SUBL** rule.

Examples of use of the command

```
NuITP> apply subl to 0.1 .
```

```
NuITP> apply subl! to 0.1 .
```

```
NuITP> apply subl* to 0.1 .
```

```
NuITP> apply subl{MY-STRAT} to 0.1 .
```

Requirements of the command

The condition Γ of the multiclause $\Gamma \to \Delta$ contains an equality $\bar{x} = u$ such that: (i) $\bar{x}$ is either a variable or a fresh (Skolem) constant; (ii) $\bar{x}$ does not appear in u; (iii) the least sort of u is lesser or equal to the sort of $\bar{x}$; (iv) u is not a Σ_1-term; and (v) the rest of the condition does not contain any Σ_1-equality (see Section 1.3).

4.5.5 Substitution Right (SUBR)

The substitution right (**SUBR**) rule substitutes equations of the form $x = u$ from the right-hand side of the multiclause, with x a variable, or $\overline{x} = u$, with $\overline{x}$ a (universal) Skolem constant.

When x is a variable, the inference rule is as follows.

$$\frac{[\overline{X},\mathcal{E},H] \Vdash \Gamma \rightarrow x = u \qquad [\overline{X},\mathcal{E},H] \Vdash (\Gamma \rightarrow \Lambda)\{x \mapsto u\}}{[\overline{X},\mathcal{E},H] \Vdash \Gamma \rightarrow \Lambda \wedge x = u}$$

provided (i) $\Lambda \neq \top$, (ii) variable x has sort s and $ls(u) \leqslant s$, and (iii) $x \notin vars(u)$. (i) avoids looping, and (ii) makes $\{x \mapsto u\}$ an order-sorted substitution. Cases $x = u$ and $u = x$ are both covered.

See [16] for the case in which $\overline{x}$ is a fresh constant.

Syntax of the command

apply subr[! | * | {<*strat-id*>}] to <*goal-id*> .

where <*strat-id*> is the identifier of the strategy to be applied on the generated subgoals, and <*goal-id*> is the identifier of the goal that we want to simplify with the **SUBR** rule.

Examples of use of the command

```
NuITP> apply subr to 0.1 .
```

```
NuITP> apply subr! to 0.1 .
```

```
NuITP> apply subr* to 0.1 .
```

```
NuITP> apply subr{MY-STRAT} to 0.1 .
```

Requirements of the command

The right-hand side Λ of the multiclause $\Gamma \rightarrow \Lambda$ contains an equality $\overline{x} = u$ such that: (i) $\overline{x}$ is either a variable or a fresh (Skolem) constant; (ii) $\overline{x}$ does not appear in u; (iii) the least sort of u is lesser than or equal to the sort of $\overline{x}$; (iv) u is not a Σ_1-term; and (v) the rest of the multiclause's right-hand side is not empty.

4.5.6 Narrowing Simplification (NS)

The **NS** rule performs one step of symbolic evaluation on a term $f(\vec{v})$ of an equality of the form $f(\vec{v}) = u$ appearing anywhere in a goal $\Gamma \rightarrow \Lambda$. The requirements for applying **NS** are that: (1) f is a non-constructor function symbol, so that its defining equations are sufficiently complete, (2) the argument subterms $\vec{v}$ are constructor terms, (3) term u belongs to the FVP subtheory $\mathcal{E}_1$ of the module's theory $\mathcal{E}$, and (4) the equations defining f, oriented as rewrite rules, have the form $\{[i] : f(\vec{u_i}) \rightarrow r_i \textit{ if } \Gamma_i\}_{i \in I}$, where the $\vec{u_i}$ are construtor terms and the r_i are terms in $\mathcal{E}_1$.

If $f(\vec{v})$ were to be a ground term, by sufficient completeness of f one of the above rules defining f would apply to $f(\vec{v})$. The important advantage of the **NS** rule is that it allows us to symbolically evaluate $f(\vec{v})$ when its arguments $\vec{v}$ are not ground terms, i.e., have variables. This symbolic evaluation process is called *narrowing* (see Section 1.1). The point is that when $f(\vec{v})$ has variables it may not *match* one of the lefthand sides $f(\vec{u_i})$ modulo the module's axioms B with a matching substitution θ (so that it can be rewritten to $r_i\theta$ with rule $[i]$ if $\Gamma_i\theta$ holds); but $f(\vec{v})$ and $f(\vec{u_i})$ can *unify* modulo B with some unifier substitution α solving the equation $f(\vec{v}) = f(\vec{u_i})$, i.e., such that $f(\vec{v})\alpha =_B f(\vec{u_i})\alpha$. In this way, the term $f(\vec{v})\alpha$ can be rewritten modulo B in one step to the term $r_i\alpha$ under the assumption that the condition $\Gamma_i\alpha$ holds. We then say that $f(\vec{v})$ *narrows* in one step to $r_i\alpha$ with rule $[i] : f(\vec{u_i}) \rightarrow r_i \textit{ if } \Gamma_i$ and B-unifier α under the assumption $\Gamma_i\alpha$.

To simplify the exposition, we give below the inference rule for the case when the argument subterms $\vec{v}$ have no Skolem constants. The more general **NS** rule dropping this requirement is presented in [16, §4.1].

$$\frac{\{[\overline{X},\mathcal{E},H] \Vdash (\Gamma_i, (\Gamma \rightarrow \Lambda)[r_i = u]_p)\alpha_{i,j}\}_{i \in I_0}^{j \in J_i}}{[\overline{X},\mathcal{E},H] \Vdash (\Gamma \rightarrow \Lambda)[f(\vec{v}) = u]_p}$$

where $I_0 \subseteq I$ is the subset of rule labels of the rules defining f such that there is at least one B-unifier of $f(\vec{v})$ and $f(\vec{u_i})$, and $\{\alpha_{i,j}\}_{j \in J_i}$ denotes a complete set of B-unifiers of the equation $f(\vec{v}) = f(\vec{u_i})$. That is, for each $i \in I_0$ and B-unifier $\alpha_{i,j}$ we get an instance subgoal $(\Gamma_i, (\Gamma \to \Lambda)[r_i = u]_p)\alpha_{i,j}$, where: (i) $f(\vec{v})\alpha_{i,j}$ has been rewritten to $r_i\alpha_{i,j}$, and (ii) the condition $\Gamma_i\alpha_{i,j}$ has been added to the subgoal's condition.

Why are the r_i required to be $\mathcal{E}_1$-terms? Because then the resulting subgoals all have equations of the form $(r_i = u)\alpha_{i,j}$, which are $\mathcal{E}_1$-equations, to which either the **CVUL** or **CVUFR** simplification rules can be applied, depending on where $f(\vec{v}) = u$ occurs in the original goal. In this way, the resulting subgoals are ripe for a subsequent simplification step by constructor variant unification. Therefore, the application of **NS** tends to be more effective when $f(\vec{v}) = u$ appears in Γ.

Let us see the power of the **NS** on an example.

Example 3 *In this example, we want to specify a type of card decks and some operations on them. We will assume that the deck is represented as a list of cards, so we have sorts Boolean, Card, NeList, and List. For boolean values and lists, we have the usual declarations:*

```
sorts Card NeList List Boolean .
subsort Card < NeList < List .

op True : -> Boolean [ctor rpo 0] .
op False : -> Boolean [ctor rpo 1] .

op nil : -> List [ctor rpo 2] .
op __ : List List -> List [assoc id: nil rpo 5] .
op __ : NeList NeList -> NeList [ctor ditto] .
```

To simplify the example a bit, we assume that cards can just either be of color red or black, that is, we have just two constructors for the Card type.

```
op black : -> Card [ctor rpo 3] .
op red : -> Card [ctor rpo 4] .
```

Then, given variables C, $C1$, $C2$ and $C3$ of type Card, and $L1$, $L2$ and $L3$ of type List, we declare operations paired, that checks whether two cards compose a pair of different cards, and shuffle, that checks whether the list of cards given as third argument is the result of shuffling the card lists given as first two arguments.

```
op paired : Card Card -> Boolean [rpo 6] .
eq paired(red, black) = True [variant] .
eq paired(black, red) = True [variant] .
eq paired(C, C) = False [variant] .

op shuffle : List List List -> Boolean [rpo 10] .
eq shuffle(nil, nil, nil) = True .
eq shuffle(nil, nil, C3 L3) = False .
eq shuffle(C1 L1, L2, nil) = False .
ceq shuffle(C1 L1, nil, C3 L3) = False if paired(C1, C3) = True .
ceq shuffle(C1 L1, nil, C3 L3) = shuffle(L1, nil, L3) if paired(C1, C3) = False .
eq shuffle(L1, C2 L2, nil) = False .
ceq shuffle(nil, C2 L2, C3 L3) = False if paired(C2, C3) = True .
ceq shuffle(nil, C2 L2, C3 L3) = shuffle(nil, L2, L3) if paired(C2, C3) = False .
ceq shuffle(C1 L1, C2 L2, C3 L3) = True
  if paired(C1, C3) = False /\ shuffle(L1, C2 L2, L3) = True .
ceq shuffle(C1 L1, C2 L2, C3 L3) = True
  if paired(C2, C3) = False /\ shuffle(C1 L1, L2, L3) = True .
ceq shuffle(C1 L1, C2 L2, C3 L3) = False
  if paired(C1, C3) = True /\ paired(C2, C3) = True .
ceq shuffle(C1 L1, C2 L2, C3 L3) = False
  if shuffle(L1, C2 L2, L3) = False /\ shuffle(C1 L1, L2, L3) = False .
ceq shuffle(C1 L1, C2 L2, C3 L3) = False
  if paired(C1, C3) = True /\ shuffle(C1 L1, L2, L3) = False .
ceq shuffle(C1 L1, C2 L2, C3 L3) = False
  if paired(C2, C3) = True /\ shuffle(L1, C2 L2, L3) = False .
```

Note that the equations defining the paired predicate are declared with the `variant` *attribute, which is key to perform constructor-based variant simplification. The definition of the shuffle operation has many equations, but notice that they are very simple, and most of them are symmetric, considering the cases nil for each of the two first arguments, and C L for each of these arguments matching and not matching C with the first card in the third argument list. Note that the paired operator is the negation of an equality predicate on cards, that is, paired(C1, C2) = False iff C1 = C2.*

Suppose now that we have the goal shuffle(nil, nil, L:NeList) = False. To prove it we could consider the ***CAS*** *or* ***GSI*** *rules. But, actually, the* ***NS*** *rule can handle it quite well. The goal can be proved just by one single command:*

```
> apply ns! to 0 .
```

The application of the ***NS*** *rule produces three equal subgoals False = False, which are discarded by equality predicate simplification.*

The goal shuffle(L1:NeList, L2:NeList, C:Card) = False can also be proved with a single application of the ***NS*** *rule with the bang. However, in this case, 99 subgoals were generated, all automatically discarded by the* ***EPS*** *rule.*

Syntax of the command

`apply ns`[`!` | `*` | `{`<*strat-id*>`}`] `to` <*goal-id*> [`on` <*subterm*>] `.`

where <*strat-id*> is the identifier of the strategy to be applied on the generated subgoals, and <*goal-id*> is the identifier of the goal that we want to simplify with the **NS** rule.

Examples of use of the command

```
NuITP> apply ns to 0.1 .
```

```
NuITP> apply ns* to 0.1 .
```

```
NuITP> apply ns! to 0.1 on rev($1:NeList $2:Elt) .
```

```
NuITP> apply ns{MY-STRAT} to 0.1 on rev($1:NeList $2:Elt) .
```

Requirements of the command

The goal's multiclause contains an equality $f(\vec{v}) = u$ such that: (i) $f(\vec{v})$ is the *narrowex*, with f a non-constructor function symbol in Σ; (ii) $f(\vec{v})$ is also not a Σ_1 term; (iii) the terms $\vec{v}$ are constructor terms; and (iv) u is a Σ_1 term (see Section 1.3).

4.5.7 Clause Subsumption (CS)

Figuring out when one clause subsumes another may help simplifying our goals.

$$\frac{[\overline{X},\mathcal{E},H \cup \{\Gamma \to \Delta\}] \Vdash \Gamma' \to (ESC)\theta}{[\overline{X},\mathcal{E},H \cup \{\Gamma \to \Delta\}] \Vdash \Gamma' \to \Lambda'}$$

where θ is a matching substitution such that $\Gamma' \to \Lambda' =_{B_0^=} (\Gamma, ES \to \Delta, ES' \wedge ESC)\theta$. That is, the hypothesis $\Gamma \to \Delta$ is extended to the pattern $\Gamma, ES \to \Delta, ES' \wedge ESC$ for pattern matching multiclauses, where the variables ES and ES' range over *equation sets* (with ES, appearing on the left side, understood as a conjunction, and ES', appearing on the right side, understood as a disjunction), and the variable ESC ranges over *equation set conjunctions* (with such equation sets understood as disjunctions).

Note that this rule can be fully automated: if a substitution θ is found such that the goal matches the extended pattern of some hypothesis $\Gamma \to \Delta$ in the goal's inductive theory, then the goal can be automatically simplified.

Let us illustrate the **CS** rule by means of a simple example. Suppose that the inductive theory involves the Peano natural numbers with 0, s, $+$, $*$ and $>$, the goal formula is:

$$\Gamma' \to \Delta' \equiv \mathit{true} = s(n) > m, \mathit{true} = m > \overline{z}, m*m = m \to s(n) > \overline{z} = \mathit{true} \wedge (n*n = n, n > m = \mathit{true})$$

and we have the (non-executable in this case) hypothesis:

$$\Gamma \to \Delta \equiv x > y = \mathit{true}, y > \overline{z} = \mathit{true} \to x > \overline{z} = \mathit{true}.$$

Then, the **CS** rule can be applied with matching substitution $\theta = \{x \mapsto s(n), y \mapsto m, \mathit{ES} \mapsto (m*m = m), \mathit{ES}' \mapsto \emptyset, \mathit{ESC} \mapsto (n*n = n, n > m = \mathit{true})\}$, and yields the resulting subgoal:

$$\Gamma' \to \theta(\mathit{ESC}) \equiv \mathit{true} = s(n) > m, \mathit{true} = m > \overline{z}, m*m = m \to n*n = n, n > m = \mathit{true}.$$

Syntax of the command

apply cs$\big[$! | * | {<*strat-id*>}$\big]$ to <*goal-id*> .

where <*strat-id*> is the identifier of the strategy to be applied on the generated subgoals, and <*goal-id*> is the identifier of the goal that we want to simplify with the **CS** rule.

Examples of use of the command

```
NuITP> apply cs to 0.1 .
```

```
NuITP> apply cs! to 0.1 .
```

```
NuITP> apply cs* to 0.1 .
```

```
NuITP> apply cs{MY-STRAT} to 0.1 .
```

Requirements of the command

The set of (executable and non-executable) hypotheses of the goal contains a hypothesis that subsumes (part of) the goal's multiclause.

4.5.8 Inductive Congruence Closure (ICC)

The **ICC** rule is a *modus ponens* type of rule that tries to discharge a goal $[\overline{X}, \mathcal{E}, H] \Vdash \Gamma \to \Lambda$ by assuming its condition Γ to prove its conclusion Λ. For this purpose, the goal's variables are temporarily converted into Skolem constants, so that we use the ground conjunction $\overline{\Gamma}$ to try to prove the ground formula $\overline{\Lambda}$. The "**I**" in **ICC** means that the attempt to prove $\overline{\Lambda}$ uses $\overline{\Gamma}$ in conjunction with the induction hypotheses H and the equality predicates $\mathcal{E}^{=}$. Specifically, **ICC** tries to prove $\overline{\Lambda}$ equivalent to $\top$ by *rewriting* it with $\overline{\Gamma}$, the executable hypotheses in H and $\mathcal{E}^{=}$. The problem is that the ground equations in $\overline{\Gamma}$, though orientable as terminating rules using an RPO order $\succ$ modulo the module's axioms, are generally not confluent. This problem can be solved by transforming them into equivalent convergent ground rules $cc^{\succ}_{B_0}(\overline{\Gamma})$ using an order-sorted congruence closure algorithm modulo axioms [13]. This will already succeed in some cases, but **ICC** introduces the following important improvement.

The equations associated to the rules $cc^{\succ}_{B_0}(\overline{\Gamma})$ are *simplified* by the executable hypotheses in H and by $\mathcal{E}^{=}$ and *inter-reduced*, that is, they are simplified with each other: for each $(l \to r) \in cc^{\succ}_{B_0}(\overline{\Gamma})$ the ground equation $l = r$ is simplified by the executable hypotheses in H, $\mathcal{E}^{=}$ *and* all other rules in $cc^{\succ}_{B_0}(\overline{\Gamma})$. In this way, two things can happen: (i) the equality predicates may show $cc^{\succ}_{B_0}(\overline{\Gamma})$ (and therefore $\overline{\Gamma}$) unsatisfiable, thus proving the goal, or (ii) since the equality predicates can transform an equation into a conjunction or a disjunction of other equations, the equations associated to $cc^{\succ}_{B_0}(\overline{\Gamma})$ are transformed by this inter-reduction process into a disjunction of conjunctions of equations that we can denote as $\overline{\Gamma}^{\sharp} = \bigvee_{i \in I} \overline{\Gamma}^{\sharp}_i$. Since we have the Boolean equivalence $(A \vee B) \Rightarrow C \equiv (A \Rightarrow C) \wedge (B \Rightarrow C)$, this gives us a piecemeal method to discharge $\overline{\Lambda}$, namely, to try to rewrite it to $\top$ with the executable hypotheses in H, $\mathcal{E}^{=}$ and each $\vec{\overline{\Gamma}}^{\sharp}_i$, $i \in I$, where $\vec{\overline{\Gamma}}^{\sharp}_i$ are the equations in $\overline{\Gamma}^{\sharp}_i$ oriented as rules with $\succ$. This is precisely what the **ICC** rule does:

$$\frac{\{[\overline{X},\mathcal{E},H] \Vdash \Gamma^{\sharp}_i \to \Lambda^{\sharp}_i\}_{i\in I}}{[\overline{X},\mathcal{E},H] \Vdash \Gamma \to \Lambda}$$

where: (i) by definition, $\overline{\Lambda}^{\sharp}_i \in \overline{\Lambda}!_{\vec{\mathcal{E}}^{=}_{\overline{X\cup Y}_U} \cup \vec{H}^{+}_{e_U} \cup \vec{\overline{\Gamma}}^{\sharp}_i}$, and we always pick $\overline{\Lambda}^{\sharp}_i = \top$ if $\top \in \overline{\Lambda}!_{\vec{\mathcal{E}}^{=}_{\overline{X\cup Y}_U} \cup \vec{H}^{+}_{e_U} \cup \vec{\overline{\Gamma}}^{\sharp}_i}$; (ii) $\Gamma^{\sharp}_i \to \Lambda^{\sharp}_i$ is obtained from $\overline{\Gamma}^{\sharp}_i \to \overline{\Lambda}^{\sharp}_i$ by converting back the Skolem constants associated to the variables of $\Gamma \to \Lambda$ into those same variables, and (iii) the case $\overline{\Gamma}^{\sharp} = \bot$ is the case when there are no conjunctions in the disjunctive normal form $\overline{\Gamma}^{\sharp}$, i.e., $I = \varnothing$, so that, by convention, the notation $\{[\overline{X},\mathcal{E},H] \Vdash \Gamma^{\sharp}_i \to \Lambda^{\sharp}_i\}_{i\in\varnothing}$ then denotes $\top$, because, since Γ has been shown inductively unsatisfiable, the goal is proved. Note that when $\overline{\Gamma}^{\sharp} = \top$, $|I| = 1$, and what we get is the subgoal $[\overline{X},\mathcal{E},H] \Vdash \Lambda'$, with $\overline{\Lambda'} \in \overline{\Lambda}!_{\vec{\mathcal{E}}^{=}_{\overline{X\cup Y}_U} \cup \vec{H}^{+}_{e_U}}$.

Example 4 *(see [16, §4.1]) Let us simplify with **ICC** the following goal involving natural number addition and multiplication in Peano notation:*

$$[\varnothing,\mathcal{N},\varnothing] \Vdash x*x = s(y), y = 0 \to x + (x*x) = s(x+y) \wedge x*x = x$$

After adding new Skolem constants $\overline{x}$ and $\overline{y}$ to the signature of $\mathcal{N}$, we can linearly order its symbols in the chain:

$$* \succ + \succ \overline{x} \succ \overline{y} \succ s \succ 0.$$

*The congruence closure of the condition's ground equations $\overline{\Gamma} = \{\overline{x}*\overline{x} = s(\overline{y}), \overline{y} = 0\}$ obtained using the associated RPO order is the set of rules $\{\overline{x}*\overline{x} \to s(0), \overline{y} \to 0\}$, which, since it cannot be further simplified by the rules in $\vec{\mathcal{N}}^{=}$, there are no induction hypotheses, and is already inter-reduced, is just $\vec{\overline{\Gamma}}^{\sharp}$. In this case, $\overline{\Lambda} = \overline{x} + (\overline{x}*\overline{x}) = s(\overline{x}+\overline{y}) \wedge \overline{x}*\overline{x} = \overline{x}$, which is simplified by $\vec{\mathcal{N}}^{=} \cup \vec{\overline{\Gamma}}^{\sharp}$ (where $\vec{\mathcal{N}}^{=}$ includes the rule $x = x \to \top$) to $s(0) = \overline{x}$. Therefore, **ICC** has simplified our goal to the considerably simpler goal,*

$$[\varnothing,\mathcal{NP},\varnothing] \Vdash x*x = s(0), y = 0 \to s(0) = x.$$

Syntax of the command

```
apply icc[! | * | {<strat-id>}] to <goal-id> .
```

where <*strat-id*> is the identifier of the strategy to be applied on the generated subgoals, and <*goal-id*> is the identifier of the goal that we want to simplify with the **ICC** rule.

Examples of use of the command

```
NuITP> apply icc to 0.1 .
```

```
NuITP> apply icc! to 0.1 .
```

```
NuITP> apply icc* to 0.1 .
```

```
NuITP> apply icc{MY-STRAT} to 0.1 .
```

Requirements of the command

Rule application fails if no simplification is achieved and only the same original goal is internally computed.

4.5.9 Equality (EQ)

The `apply eq` command allows the application of a (non-orientable) hypothesis as a rewrite rule in either direction. Similarly, it allows the application of any equation in our module either from left to right, as the **EPS** rule would do, or from right to left.

Syntax of the command

`apply eq`[`!` | `*` | `{`<*strat-id*>`}`] `to` <*goal-id*> `with` <*hypothesis*> [`sub` <*substitution*>] `.`

where <*strat-id*> is the identifier of the strategy to be applied on the generated subgoals, <*goal-id*> is the identifier of the goal on which we want to apply the `EQ` rule, <*hypothesis*> is the *oriented* version of a non-executable hypothesis of the goal (which must be either an equation or a conditional equation), and <*substitution*> is an (optional and possibly partial, i.e., specified only for some variables) substitution, whose domain is a subset of the set of variables of the chosen non-executable hypothesis. If no (possibly partial) substitution is specified, the rule is attempted using the empty substitution, i.e., trying to rewrite the goal's multiclause in one step with the oriented (and possibly conditional) hypothesis as a rewrite rule. The optional partial substitution can be used both to restrict the possible applications of such a rewrite rule, and/or to instantiate those variables in the rule's righthand side or condition that do not appear in the rule's lefthand side. As usual, the `apply eq!` version of the command corresponds to applying **EQ** followed by **EPS**.

Examples of use of the command

```
NuITP> apply eq to 0.1 with N:Nat + M:Nat => M:Nat + N:Nat .
```

```
NuITP> apply eq! to 0.1 with N:Nat + M:Nat => M:Nat + N:Nat .
```

```
NuITP> apply eq* to 0.1 with N:Nat + M:Nat => M:Nat + N:Nat .
```

```
NuITP> apply eq{MY-STRAT} to 0.1 with N:Nat + M:Nat => M:Nat + N:Nat .
```

```
NuITP> apply eq to 0.1 with N:Nat + M:Nat => M:Nat + N:Nat sub N:Nat <- 0 ; M:Nat <-
s(0) .
```

```
NuITP> apply eq! to 0.1 with N:Nat + M:Nat => M:Nat + N:Nat sub N:Nat <- 0 ; M:Nat <-
s(0) .
```

```
NuITP> apply eq* to 0.1 with N:Nat + M:Nat => M:Nat + N:Nat sub N:Nat <- 0 ; M:Nat <-
s(0) .
```

```
NuITP> apply eq{MY-STRAT} to 0.1 with N:Nat + M:Nat => M:Nat + N:Nat sub N:Nat <- 0 ;
M:Nat <- s(0) .
```

See an example of use in Section 5.4.

Requirements of the command

The goal contains at least one non-executable hypothesis that is either an equation or a conditional equation that the user can manually orient as a rule in a manner that makes it applicable to the goal.

4.5.10 Decompose Equivalence (DEQ)

The Decompose Equivalence (**DEQ**) rule splits a double implication into two goals, one for each of its directions. Thus, the `apply deq` command is used to decompose an equivalence $u=v \leftrightarrow u'=v'$ into two separate goals: one proving $\Gamma, u=v \rightarrow u'=v'$ and another proving $\Gamma, u'=v' \rightarrow u=v$. To comply with the clause format, the equivalence $A \Rightarrow (B \Rightarrow C) \equiv (A \wedge B) \Rightarrow C$ is used.

$$\frac{[\overline{X},\mathcal{E},H] \Vdash \Gamma, u=v \rightarrow u'=v' \qquad [\overline{X},\mathcal{E},H] \Vdash \Gamma, u'=v' \rightarrow u=v}{[\overline{X},\mathcal{E},H] \Vdash \Gamma \rightarrow u=v \leftrightarrow u'=v'}$$

Syntax of the command

`apply deq`$\big[$`!` | `*` | `{`$<$*strat-id*$>$`}`$\big]$ `to` $<$*goal-id*$>$ `.`

Examples of use of the command

```
NuITP> apply deq* to 0 .
```

4.5.11 Clause Boolean Equality (CBEQL and CBEQR)

The clause boolean equivalence rules use the equivalence $\neg(u=b) \equiv u = not(b)$, with $b \in \{\mathbf{True}, \mathbf{False}\}$, to move boolean equalities from one side of a clause's implication to the other. The Clause Boolean Equivalence Right rule (**CBEQR**) is

$$\frac{[\overline{X},\mathcal{E},H] \Vdash \Gamma \rightarrow u=b, \Delta}{[\overline{X},\mathcal{E},H] \Vdash u=not(b), \Gamma \rightarrow \Delta}$$

and the Clause Boolean Equivalence Left rule (**CBEQL**) is

$$\frac{[\overline{X},\mathcal{E},H] \Vdash u=b, \Gamma \rightarrow \Delta}{[\overline{X},\mathcal{E},H] \Vdash \Gamma \rightarrow \Delta, u=not(b)}$$

where u is a term of sort **Bool**[20] and $b \in \{\mathbf{True}, \mathbf{False}\}$.

Examples of use of the command

```
NuITP> apply cbeqr* to 0 .
```

4.5.12 Restart (RST)

The `apply rst` command allows the *de-Skolemization* of a goal, also removing any hypothesis containing *Skolem* constants. This command is a special, automatable instance of the Lemma Enrichment command (see Section 4.6.3). It is quite useful to continue the proof effort when we have "runned out of variables" to induct on in the given subgoal, which may have even become a *ground formula*, because previous variables were Skolemized away earlier in the proof process. The `rst` command is also very useful because it makes it easy to take advantage of the fact that, in inductive theorem proving, *failures* to prove a goal, in the form of the subgoals left unproved, are an unending source of useful hints for *lemmas* to be proved. The `rst` command makes this lemma guessing easy for the user without any need for an explicit Lemma Enrichment command; it even makes lemma guessing *automatable* as part of a strategy.

Syntax of the command

`apply rst`$\big[$`!` | `*` | `{`$<$*strat-id*$>$`}`$\big]$ `to` $<$*goal-id*$>$ `.`

where $<$*strat-id*$>$ is the identifier of the strategy to be applied on the generated subgoal and $<$*goal-id*$>$ is the identifier of the goal on which we want to apply the `RST` rule.

[20]NuITP requires that the module `BOOL-NuITP` is used for the Booleans.

Examples of use of the command

```
NuITP> apply rst to 0.1 .
```

```
NuITP> apply rst! to 0.1 .
```

```
NuITP> apply rst* to 0.1 .
```

```
NuITP> apply rst{MY-STRAT} to 0.1 .
```

Requirements of the command

The goal contains at least one *Skolem* constant.

4.6 Induction commands

In the following, we show the induction commands available in the current version of NuITP. Note that all induction commands have at least three options, the simple version that applies the specified rule and two extended `!` and `*` versions that apply the rule likewise, but followed by **EPS** (`!`) or an even more powerful goal simplification strategy (`*`)[21] in order to automatically simplify the rule's resulting subgoals.

4.6.1 Generator Set Induction (GSI)

The **GSI** rule generalizes standard structural induction on constructors. In both structural induction and its **GSI** generalization, one inducts on a variable z in the goal. Let us consider the following simplified **GSI** inference rule — see [16] for the fully general version of the rule used by NuITP:

$$\frac{\{[\overline{X} \uplus \overline{Y}_i, \mathcal{E}, H \uplus H_i] \Vdash (\Gamma \to \bigwedge_{j\in J} \Delta_j)\{z \mapsto \overline{u}_i\}\}_{1\leqslant i\leqslant n}}{[\overline{X}, \mathcal{E}, H] \Vdash \Gamma \to \bigwedge_{j\in J} \Delta_j}$$

where $z \in vars(\Gamma \to \bigwedge_{j\in J} \Delta_j)$ has sort s, $\{u_1,\ldots,u_n\}$ is a B_0-generator set for s, with B_0 the non-unit axioms of the theory $\vec{\mathcal{E}}$, and Y_i are the variables of the u_i, $Y_i = vars(u_i)$, for $1 \leqslant i \leqslant n$. These variables are new variables to avoid conflicts. The induction hypotheses H_i are:

$$H_i = \{(\Gamma \to \Delta_j)\{z \to \overline{v}\} \mid v \in PST_{B_0,\leqslant s}(u_i) \wedge j \in J\}$$

where $PST_{B_0,\leqslant s}(u)$ are the proper B_0-subterms of u,

$$PST_{B_0,\leqslant s}(u_i) = \{v \in PST_{B_0}(u_i) \mid ls(v) \leqslant s\},$$

$\overline{v}$ denotes the instantiation of v by the substitution $\{y \mapsto \overline{y}\}_{y\in Y_i}$ with $\overline{y}$ a fresh Skolem constant of same sort as y, and $\overline{u}_i$ denotes the instantiation of u_i by the substitution $\{y \mapsto \overline{y}\}_{y\in Y_i}$. For example, for generator set $\{0, s(N)\}$, $s(N)$ has one proper B_0-subterm, N; and for generator set $\{0, 1, s(s(N))\}$, $s(s(N))$ has proper B_0-subterms N and $s(N)$.[22]

Let us illustrate the **GSI** rule with an example that should be familiar to many readers, but may still be helpful to illustrate the notation and use of **GSI**.

Example 5 (Borrowed from [16]) *Consider an equational theory $\mathcal{E}$ with a specification of lists of natural numbers, with sorts Nat and List. The constructors for sort List are a constant nil and an operator $\_\cdot\_ : Nat\ List \to List$. Let us assume in this theory an append operator $\_@\_ : List\ List \to List$ defined with the equations $nil @ L = L$ and $(N \cdot L) @ Q = N \cdot (L @ Q)$, where N is a variable of sort Nat and L and Q are variables of sort List. On this theory $\mathcal{E}$, let us prove the associativity of the append operator, that is, the goal:*

$$[\varnothing, \mathcal{E}, \varnothing] \Vdash (L @ P) @ Q = L @ (P @ Q).$$

[21] To learn more about simplifying strategies see Section 3.

[22] Please, see [16, Definition 2 and Remark 1] for a definition of the notion of proper B_0-subterm.

with L, P and Q variables of sort List. We can do so by applying the ***GSI*** *rule to the goal on variable L with generator set $\{nil,\ M \cdot R\}$, with M a variable of sort Nat and R a variable of sort List, i.e., by structural list induction. We get two goals, one for nil and one for $M \cdot R$:*

$$[\varnothing,\mathcal{E},\varnothing] \Vdash (nil @ P) @ Q = nil @ (P @ Q)$$

and

$$[\{\overline{M},\overline{R}\},\mathcal{E},\{(\overline{R} @ P) @ Q = \overline{R} @ (P @ Q)\}] \Vdash ((\overline{M} \cdot \overline{R}) @ P) @ Q = (\overline{M} \cdot \overline{R}) @ (P @ Q).$$

In the case of nil, $Y_1 = \varnothing$, and therefore no new constants are introduced. For the generator $M \cdot R$, $Y_2 = \{M,R\}$, $PST_{B_0,\leqslant_{List}}(M \cdot R) = \{R\}$, and H_2 is the induction hypothesis $(\overline{R} @ P) @ Q = \overline{R} @ (P @ Q)$.

It can be observed how both subgoals can easily simplify to $\top$. The first subgoal can be simplified using the equation $nil @ L = L$ on both sides of the equation, leading to $P @ Q = P @ Q$, which is finally reduced to $\top$ using the equationally-defined equality predicate rules in $\mathcal{E}^{=}$. For the second subgoal, and given the standard RPO, we have $(\overline{R} @ P) @ Q \succ \overline{R} @ (P @ Q)$, and therefore the hypothesis of that goal can be used as the rewrite rule $(\overline{R} @ P) @ Q \to \overline{R} @ (P @ Q)$. Thus, the subgoal

$$((\overline{M} \cdot \overline{R}) @ P) @ Q = (\overline{M} \cdot \overline{R}) @ (P @ Q)$$

can be rewritten, using the second equation on both sides, to

$$\overline{M} \cdot ((\overline{R} @ P) @ Q) = \overline{M} \cdot (\overline{R} @ (P @ Q)),$$

which can be rewritten

$$\overline{M} \cdot (\overline{R} @ (P @ Q)) = \overline{M} \cdot (\overline{R} @ (P @ Q)),$$

which can then be reduced to $\top$ using the rules in $\mathcal{E}^{=}$. All these simplifications are automated by the ***EPS*** *simplification rule and corresponding NuITP command.*

Additional simple examples of the use of the **GSI** rule can be found in Sections 2.2-2.9. The reader is referred to [16, §4.2] for additional details and examples on the **GSI** rule.

The generalization of the **GSI** rule to n distinct variables $z_1,\ldots,z_n$ of sorts $s_1,\ldots,s_n$, $n \geqslant 1$, among those appearing in a conjecture $\Gamma \to \bigwedge_{j \in J} \Delta_j$ is as follows:

$$\frac{\{[\overline{X} \uplus \overline{Y}_{\vec{u}},\mathcal{E},H \uplus H_{\vec{u}}] \Vdash (\Gamma \to \bigwedge_{j \in J} \Delta_j)\{z \mapsto \overline{\vec{u}}\}\}_{\vec{u} \in G_1 \times \ldots \times G_n}}{[\overline{X},\mathcal{E},H] \Vdash \Gamma \to \bigwedge_{j \in J} \Delta_j}$$

where $\vec{z}$ denotes the tuple $(z_1,\ldots,z_n)$, $\vec{s}$ denotes the tuple $(s_1,\ldots,s_n)$, for $\vec{u} = (u_1,\ldots,u_n)$, $\{\vec{z} \mapsto \overline{\vec{u}}\}$ denotes the substitution $\{z_1 \mapsto \overline{u_1},\ldots,z_n \mapsto \overline{u_n}\}$, $G_1 \times \ldots \times G_n$ is the Cartesian product of the B_0-generator sets G_i for sorts s_i, $1 \leqslant i \leqslant n$, all having fresh variables, and such that for each $i,j, 1 \leqslant i \leqslant j \leqslant n$, $vars(G_i) \cap vars(G_j) = \varnothing$, and $\overline{Y}_{\vec{u}} = vars(\vec{u})$. In the generalized form of the **GSI** command, the user can choose to apply the rule on a single variable or on several variables at once. In case the user chooses to apply the rule on several variables, the default generator sets for the sorts of the variables involved are used.

Syntax of the command

`apply gsi`$\big[$`!` | `*` | `{`*<strat-id>*`}`$\big]$ `to` *<goal-id>* $\big[$`on` *<vars>*$\big]$ $\big[$`with` *<gen-set-ids>* | *<gen-set>*$\big]$ `.`

where *<goal-id>* is the identifier of the goal on which we want to apply the **GSI** rule, *<vars>* is the list of variables on which we want to apply induction, separated by blank spaces, *<gen-set-ids>* is the list of identifiers of previously defined generator sets, also separated by blank spaces, *<gen-set>* is a set of terms separated by double semicolons that we want to use as a generator set for all ground constructor terms of the variable's sort (only in case we apply the rule on a single variable).

Examples of use of the command

```
NuITP> apply gsi to 0.1 on $1 $2 $3 .
```

```
NuITP> apply gsi! to 0.1 on $1 $2 with GEN-NAT1 GEN-NAT2 .
NuITP> apply gsi* to 0.1 on $1 with 0 ;; s(Y:Nat) .
```

```
NuITP> apply gsi{MY-STRAT} to 0.1 on $1 with 0 ;; s(Y:Nat) .
```

See examples of use of the **GSI** rule in Sections 2.2, 2.3, 2.4, 2.6, 2.8, and 5.2.

Requirements of the command

The goal's multiclause contains the variables on which we want to apply **GSI**. Also, if not explicitly given (and only in the case of one variable), generator sets for all sorts of the variables involved should have been defined previously. Furthermore, the sorts of each generator set referenced in <*gen-set-ids*> should have a direct correspondence to the sorts of the variables in <*vars*>. Finally, the generator sets for the sort of the chosen variables should be a correct generator set. Checking the correctness of the generator set is the user's responsibility.[23]

4.6.2 Narrowing Induction (NI)

The **NI** rule can be more easily understood by comparing it to a generalized version of the **NS** rule. Consider the following generalization of the **NS** rule: (i) not requiring anymore that $f(\vec{v})$ appears in an equation $f(\vec{v}) = u$ of $\Gamma \to \Lambda$: it can now appear *anywhere* in $\Gamma \to \Lambda$, and (ii) not requiring anymore that the r_i in the rules $\{[i] : f(\vec{u_i}) \to r_i \;\, \textit{if} \;\, \Gamma_i\}_{i \in I}$ defining f are $\mathcal{E}_1$-terms. The only reason for those two requirements in **NS** was to ensure that the resulting subgoals could be further simplified by either **CVUL** or **CVUFR**; but the concept of narrowing simplification does not depend on those two requirements and can be performed in the more general way specified by the following **NS** rule:

$$\frac{\{[\overline{X},\mathcal{E},H] \Vdash (\Gamma_i,(\Gamma \to \Lambda)[r_i]_p)\alpha_{i,j}\}_{i\in I_0}^{j\in J_i}}{[\overline{X},\mathcal{E},H] \Vdash (\Gamma \to \Lambda)[f(\vec{v})]_p}$$

But why is the above rule a *simplification* rule and not an *induction* rule? Because its subgoals have *no induction hypotheses* to help proving the original goal. Here is where the **NI** rule comes in. That is, we can view the above generalized **NS** rule as a "poor man's" **NI** rule, in the exact same way as the **CAS** rule is a "poor man's" **GSI** rule.

For the sake of a simpler exposition we present below a streamlined version of the **NI** rule. The reader is referred to [16, §4.2] for a more detailed version of **NI**. The streamlined version of **NI** is as follows:

$$\frac{\{[\overline{X} \uplus \overline{Y}_{i,j},\mathcal{E},H \uplus H_{i,j}] \Vdash (\Gamma_i,(\Gamma \to \bigwedge_{l\in L}\Delta_l)[r_i]_p)\overline{\alpha}_{i,j}\}_{i\in I_0}^{j\in J_i}}{[\overline{X},\mathcal{E},H] \Vdash (\Gamma \to \bigwedge_{l\in L}\Delta_l)[f(\vec{v})]_p}$$

Just as for the **CAS** and **GSI** rules, when comparing the above generalized **NS** rule and the **NI** rule, the two main differences are: (i) the presence of the Skolem constants $\overline{Y}_{i,j}$, and (ii) the presence of the induction hypotheses $H_{i,j}$. Let us explain both. The B-unifiers $\alpha_{i,j}$ of each equation $f(\vec{v}) = f(\vec{u_i})$ are

[23]The check of the correctness of the generator set can be semi-automated using Maude's Sufficient Completeness Checker (SCC)). This checking can be made automatically when each of the constructor terms in the generator set has no repeated variables using Maude's SCC tool [10]. Alternatively, NuITP itself can be used for this purpose using the fact that a generator set based on the constructors of the given sort (which corresponds to standard structural induction on the constructors) *is* always a correct generator set. So one can use structural induction to prove the correctness of another generator set. For example, for the sort `Nat`, the generator set `0 ;; s(N:Nat)` is the correct-by-construction structural induction generator set. We can then use it to prove by induction that the generator set `0 ;; s(0) ;; s(s(N:Nat))` is also a correct generator set for `Nat`. The idea of the proof is quite simple. Let us illustrate it with the above example. We would define a module including just the constructors and `BOOL-OPS`, as well as the newly-defined predicate:

```
    op Nat :  Nat -> Bool .
```

defined by three equations:

```
    eq Nat(0) = true .
    eq Nat(s(0)) = true .
    eq Nat(s(s(N:Nat))) = true .
```

Then, the generator set `0 ;; s(0) ;; s(s(N:Nat))` will be correct if we can prove the goal `Nat(M:Nat) = true` in NuITP by structural induction. The automatic method based on invoking the SCC tool uses a similar idea: we would just check that the above definition of the `Nat` predicate is sufficiently complete using the SCC.

substitutions mapping each variable x in its domain to a term $\alpha_{i,j}(x)$. The set $Y_{i,j}$, called the *range* of $\alpha_{i,j}$, is the set of all variables appearing in the $\alpha_{i,j}(x)$. The variables $Y_{i,j}$, are always assumed *fresh*, i.e., they do not appear anywhere else. $\overline{Y}_{i,j}$ are the Skolem constants obtained by converting each $y \in Y_{i,j}$ into a Skolem constant $\overline{y} \in \overline{Y}_{i,j}$ of the same sort. Likewise, $\overline{\alpha}_{i,j}$ denotes the composed substitution $\overline{\alpha}_{i,j} = \alpha_{i,j}\{y \mapsto \overline{y}\}_{y \in Y_{i,j}}$. We explain the induction hypotheses $H_{i,j}$ when the position p occurs in some disjunction Δ_k, and refer the reader to [16, §4.2] for the definition of $H_{i,j}$ when p occurs in the condition Γ. By definition, $H_{i,j} = \{(\Gamma \to \Delta_k)\overline{\gamma} \mid f(\vec{w}) \in SSC([i]) \wedge f(\vec{w})\alpha_{i,j} =_{B_0} f(\vec{v})\gamma\}$, where $\overline{\gamma} = \gamma\{y \mapsto \overline{y}\}_{y \in Y_{i,j}}$, with B_0 the non-unit axioms in B. Intuitively, when we symbolically evaluate $f(\vec{v})$ in one step to $r_i\overline{\alpha}_{i,j}$ by narrowing with rule $[i]$, each such $f(\vec{v})\overline{\gamma}$ will be a function subcall in either $r_i\overline{\alpha}_{i,j}$ or in the instantiated rule's condition $\Gamma_i\overline{\alpha}_{i,j}$. $H_{i,j}$ allows us to induct on such function subcalls.

Example 6 *(see [16, §4.2]). Let Ω be a signature of constructors for non-empty lists with sorts Elt and List, a subsort inclusion Elt < List, and an associative (A) list concatenation operator $\_\cdot\_ : List\ List \to List$. Define a convergent and sufficiently complete list reverse function $rev : List \to List$ by the equations oriented as rules, $[1] : rev(x) \to x$ (which has no subcalls), and $[2] : rev(x \cdot L) \to rev(L) \cdot x$, with $SSC([2]) = \{rev(L)\}$, where x,x',y,y' have sort Elt, and L,P,Q have sort List.*

We would like to prove the inductive lemma:

$$rev(Q \cdot y) = y \cdot rev(Q).$$

*We apply the **NI** rule to the conjecture's lefthand side. Rule $[1]$ has no A-unifiers. Rule $[2]$ has two most general A-unifiers, namely, $\alpha_{2,1} = \{x \mapsto x',\ Q \mapsto x',\ y \mapsto y',\ L \mapsto y'\}$, with $Y_{2,1} = \{x',\ y'\}$, and $\alpha_{2,2} = \{x \mapsto x', y \mapsto y', L \mapsto P \cdot y', Q \mapsto x' \cdot P\}$, with $Y_{2,1} = \{x', y', P\}$.*
With $\alpha_{2,1}$, since for the subcall $rev(L)$ we get the instance $rev(L)\alpha_{2,1} = rev(y')$, which does not *match our focus term $rev(Q \cdot y)$ as an instance modulo A, $H_{2,1} = \varnothing$, and we just get the subgoal $rev(y') \cdot x' = y' \cdot rev(x')$, which simplifies to $\top$.*
With $\alpha_{2,2}$, the subcall $rev(L)$ yields the instance $rev(L)\alpha_{2,2} = rev(P \cdot y')$, which does *match or focus term $rev(Q \cdot y)$ as an instance modulo A with substitution $\gamma = \{Q \mapsto P, y \mapsto y'\}$. Therefore, $\overline{\gamma} = \{Q \mapsto \overline{P}, y \mapsto \overline{y'}\}$ and we get $H_{2,2} = \{rev(\overline{P} \cdot \overline{y'}) = \overline{y'} \cdot rev(\overline{P})\}$ and the subgoal $rev(\overline{P} \cdot \overline{y'}) \cdot \overline{x'} = \overline{y'} \cdot rev(\overline{x'} \cdot \overline{P})$, which simplifies to $rev(\overline{P} \cdot \overline{y'}) \cdot \overline{x'} = \overline{y'} \cdot rev(\overline{P}) \cdot \overline{x'}$, and can, using $H_{2,2}$, be further simplified to $\top$ by means of the **EPS** simplification rule. This proves the lemma by a single application of **NI** followed by **EPS**.*

Syntax of the command

apply ni[! | * | {<*strat-id*>}] to <*goal-id*> on <*subterm*> .

where <*strat-id*> is the identifier of the strategy to be applied on the generated subgoals, <*goal-id*> is the identifier of the goal to which we want to apply the **NI** rule, and <*subterm*> is a subterm of the form $f(\vec{v})$ appearing in the clause of such goal.

Examples of use of the command

```
NuITP> apply ni to 0.1 on rev($1:NeList $2:Elt) .
```

```
NuITP> apply ni* to 0.1 on rev($1:NeList $2:Elt) .
```

See examples of use in Sections 5.3.

Requirements of the command

The goal's multiclause contains the specified subterm $f(\vec{v})$ such that: (i) $f(\vec{v})$ is the *narrowex*, with f a non-constructor function symbol in Σ; (ii) $f(\vec{v})$ does not contain any Skolem constants; and (iii) the terms $\vec{v}$ are all constructor terms.

4.6.3 Lemma Enrichment (LE)

This rule allows us to add a new lemma to our current goal.

$$\frac{[\overline{X}_0,\mathcal{E},H_0] \Vdash \Gamma' \to \bigwedge_{j\in J}\Delta'_j \qquad [\overline{X},\mathcal{E},H] \Vdash H_0 \qquad [\overline{X},\mathcal{E},(H \uplus \{\Gamma' \to \Delta'_j\}_{j\in J})_{simp}] \Vdash \Gamma \to \Lambda}{[\overline{X},\mathcal{E},H] \Vdash \Gamma \to \Lambda}$$

where $\emptyset \subseteq \overline{X}_0 \subseteq \overline{X}$. Common cases of use include either: (a) $\overline{X}_0 = H_0 = \emptyset$, or (b) $\overline{X}_0 = \overline{X}$ and $H_0 = H$.

Syntax of the command

apply le$\big[$! | * | {<*strat-id*>}$\big]$ to <*goal-id*> with <*multiclause*> .

where <*strat-id*> is the identifier of the strategy to be applied on the generated subgoals, <*goal-id*> is the identifier of the goal on which we want to apply the **LE** rule, and <*multiclause*> is the (possibly conditional) multiclause that we want to introduce as a new lemma in our proof.

Examples of use of the command

```
NuITP> apply le to 0.1 with N:Nat + M:Nat = M:Nat + N:Nat .
```

```
NuITP> apply le! to 0.1 with N:Nat + M:Nat = M:Nat + N:Nat .
```

```
NuITP> apply le* to 0.1 with N:Nat + M:Nat = M:Nat + N:Nat .
```

```
NuITP> apply le{MY-STRAT} to 0.1 with N:Nat + M:Nat = M:Nat + N:Nat .
```

See an example of use in Section 5.4.

4.6.4 Split (SP)

If we can prove a disjuntion, the **SPLIT** rule allows us to consider each of the disjunts as assumptions for the goal. This may be convenient, for example, in cases in which a function is defined using conditional equations (or if-then-elses) considering complementary cases.

$$\frac{\{[\overline{X},\mathcal{E},H] \Vdash \Gamma_i\theta,\Gamma \to \Lambda\}_{i\in I} \qquad [\overline{X},\mathcal{E},H] \Vdash \mathit{cnf}(\bigvee_{i\in I}\Gamma_i)}{[\overline{X},\mathcal{E},H] \Vdash \Gamma \to \Lambda}$$

where $vars((\bigvee_{i\in I}\Gamma_i)\theta) \subseteq vars(\Gamma \to \Lambda)$, and $\mathit{cnf}(\bigvee_{i\in I}\Gamma_i)$ denotes the conjunctive normal form of $\bigvee_{i\in I}\Gamma_i$, which is a multiclause of the general form Λ'.

Two very common special cases are: (i) $\Gamma_i = (u_i = v_i)$, $i \in I$, and (ii) the even simpler subcase of a disjunction $u = \mathit{true} \vee u = \mathit{false}$, where u is a term of sort *Bool* and θ is the identity substitution.

Syntax of the command

apply sp$\big[$! | * | {<*strat-id*>}$\big]$ to <*goal-id*> with <*disjunction*> $\big[$sub <*substitution*>$\big]$.

where <*strat-id*> is the identifier of the strategy to be applied on the generated subgoals, <*goal-id*> is the identifier of the goal on which we want to apply the **SP** rule, <*disjunction*> is a disjunction used to split the goal, and <*substitution*> is a substitution whose domain is the set of variables of the disjunction.

Examples of use of the command

```
NuITP> apply sp to 0.1 with (N:Nat + s(0) > 0 = true) \/ (N:Nat + s(0) <= 0 true) .
```

```
NuITP> apply sp! to 0.1 with (N:Nat + s(0) > 0 = true) \/ (N:Nat +s(0) <= 0 = true) .
NuITP> apply sp* to 0.1 with (N:Nat + M:Nat > 0 = true) \/ (N:Nat + M:Nat <= 0 = true)
sub M:Nat <- s(0) .
```

```
NuITP> apply sp{MY-STRAT}* to 0.1 with (N:Nat + M:Nat > 0 = true) \/ (N:Nat + M:Nat <=
0 = true) sub M:Nat <- s(0) .
```

Requirements of the command

The range of <*substitution*> is contained in the set of variables of the goal's multiclause.

4.6.5 Case (CAS), Grounding (GND), and Unit-Free Downgrade (UFree)

CAS is a well-known rule in inductive theorem proving. It can be seen as a weaker version of the **GSI** rule, since subgoals are generated for each of the generators of the sort of the variable on which the rule is applied, but without adding the corresponding hypotheses.

CAS can be applied on a variable or on a Skolem constant. To simplify the exposition we discuss here the version of the inference rule when it is applied on a variable z:

$$\frac{\{[\overline{X},\mathcal{E},H] \Vdash (\Gamma \to \Lambda)\{z \mapsto u_i\}\}_{1 \leqslant i \leqslant n}}{[\overline{X},\mathcal{E},H] \Vdash \Gamma \to \Lambda}$$

where $z \in vars(\Gamma \to \Lambda)$ has sort s and $\{u_1,...,u_n\}$ is a B_0-generator set for sort s, with the u_i, for $1 \leqslant i \leqslant n$, having fresh variables.

The reader is referred to [16, §4.2] for a full account of the **CAS** rule implemented in NuITP.

Sometimes it may be useful to repeatedly apply **CAS** on different variables. This may be particularly useful for variables having a *finite* sort, whose generator set is a set of constants. For example, for variables of sort *Bool* generated by constructors *true* and *false*. The **apply gnd** command applies repeatedly the **CAS** rule on each of the variables of a finite sort of this type, leading to a subgoal for each of the possible combinations of generator instantiations of these variables.

In a way similar to the **apply gnd** command, the **apply ufree** command is syntactic sugar for a specific version of CAS in which all variables of the clause of sorts that have constructors with the unit axiom (e.g., sort *List*) are downgraded by either taking the value of the identity element or a fresh variable of their immediately subsort (in our example, sort *NeList*, which is free of the identity axiom). The command automates this downgrade of sorts by first automatically inferring the minimal generator set possible but also applying it to all the variables of the goal that have sorts with constructors *not* free of the axiom, saving the user to manually apply **CAS** a minimum of 2^n times, with n being the number of variables affected.

Syntax of the commands

`apply cas`[`!` | `*` | `{`<*strat-id*>`}`] `to` <*goal-id*> `on` <*vars*> [`with` <*gen-set-ids* `|` *gen-set*>] `.`

`apply gnd`[`!` | `*` | `{`<*strat-id*>`}`] `to` <*goal-id*> `.`

`apply ufree`[`!` | `*` | `{`<*strat-id*>`}`] `to` <*goal-id*> `.`

where <*strat-id*> is the identifier of the strategy to be applied on the generated subgoals, <*goal-id*> is the identifier of the goal to which we want to apply the rule, <*vars*> is the list of variables or *Skolem* constants on which we want to apply cases, separated by blank spaces, <*gen-set-ids*> is the list of identifiers of previously defined generator sets, also separated by blank spaces, and <*gen-set*> is a set of generator terms separated by double semicolons that, only for the case of applying the rule on a single variable or *Skolem* constant, generate all the ground constructor terms of the variable's sort (modulo the axioms holding on constructors) and is used to use to generate the different cases. Note that the **apply gnd** command will operate on all the variables of sorts whose generator sets are sets of constants (no need to specify the variables) and the default generator sets will be used in all cases (it is not possible to specify alternative generator sets).

Examples of use of the commands

```
NuITP> apply cas to 0.1 on $1 $2 $3 .
```

```
NuITP> apply cas! to 0.1 on $1 $2 with GEN-NAT1 GEN-NAT2 .
```

```
NuITP> apply cas* to 0.1 on $1 with 0 ;; s Y:Nat .
```

```
NuITP> apply cas{MY-STRAT} to 0.1 on $1 with 0 ;; s Y:Nat .
```

```
NuITP> apply gnd! to 0.1 .
```

```
NuITP> apply ufree* to 0.1 .
```

Requirements of the command

The goal's multiclause contains the variable on which we want to apply the **CAS** rule. Same as for the `apply gsi` command, if not explicitly given (and only in the case of one variable), generator sets for all sorts of the variables involved should have been defined previously. Furthermore, the sorts of each generator set referenced in <*gen-set-ids*> should have a direct correspondence to the sorts of the variables in <*vars*>. Finally, the generator sets for the sort of the chosen variables should be a correct generator set. Checking the correctness of the generator set is the user's responsibility. As already mentioned, checking the correctness of the generator set is the user's responsibility.

4.6.6 Variable Abstraction (VA)

The **VA** rule allows us to abstract away a subterm.

$$\frac{[\overline{X},\mathcal{E},H] \Vdash u = v[z]_p, z = w, \Gamma \to \Lambda}{[\overline{X},\mathcal{E},H] \Vdash u = v[w]_p, \Gamma \to \Lambda}$$

where z is a fresh variable whose sort is the least sort of subterm w of v at position p. **VA** is particularly effective when the equation $u = v[z]_p$ resulting from the abstraction is a Σ_1-equation (this may require several applications of **VA**), since then it can be unified away by **CVUL**.

Syntax of the command

apply va[! | * | {<*strat-id*>}] to <*goal-id*> on <*term*> .

where <*strat-id*> is the identifier of the strategy to be applied on the generated subgoals, and <*goal-id*> is the identifier of the goal in which we want to apply the **VA** rule and <*term*> is a (sub)term of the condition of such goal.

Examples of use of the command

```
NuITP> apply va to 0.1 on $1:NzNat * $2:NzNat .
```

```
NuITP> apply va! to 0.1 on $1:NzNat * $2:NzNat .
```

```
NuITP> apply va* to 0.1 on $1:NzNat * $2:NzNat .
NuITP> apply va{MY-STRAT} to 0.1 on $1:NzNat * $2:NzNat .
```

Requirements of the command

The goal's multiclause contains an equality $u = v$ in its condition such that: (i) u is a Σ_1 term; (ii) v is the term we provide as argument; and (iii) v is not a Σ_1 term.

4.6.7 Cut (CUT)

The **CUT** rule can be viewed as a generalized *modus ponens* rule, since when $\Gamma \equiv \top$ it actually becomes *modus ponens*. Given Γ' is a conjunction of equalities, with $vars(\Gamma') \subseteq vars(\Gamma \to \Lambda)$, the **CUT** rule is as follows.

$$\frac{[\overline{X},\mathcal{E},H] \Vdash \Gamma \to \Gamma' \qquad [\overline{X},\mathcal{E},H] \Vdash \Gamma,\Gamma' \to \Lambda}{[\overline{X},\mathcal{E},H] \Vdash \Gamma \to \Lambda}$$

The user has complete freedom to choose a suitable Γ' that will help in proving the original goal.

Syntax of the command

apply cut[! | * | {<*strat-id*>}] to <*goal-id*> with <*term*> .

where <*strat-id*> is the identifier of the strategy to be applied on the generated subgoals, <*goal-id*> is the identifier of the goal in which we want to apply the **CUT** rule, and <*term*> is either an equality of a conjunction of equalities.

Examples of use of the command

```
NuITP> apply cut to 0 with ($1:Nat + s(0) > 0 = true) .
```

```
NuITP> apply cut! to 0 with ($1:Nat + s(0) > 0 = true) .
```

```
NuITP> apply cut* to 0 with ($1:Nat + s(0) > 0 = true) /\ ($2:Nat + s(0) > 0 = true) .
```

```
NuITP> apply cut{MY-STRAT} to 0 with ($1:Nat + s(0) > 0 = true) /\ ($2:Nat + s(0) > 0
= true) .
```

It is worth noting that the **CUT** rule works very well in combination with the **CS** rule, as it allows users to introduce new equalities in the goal's condition and hopefully match a non-executable hypothesis, which will subsume the clause of the goal and thus prove it. However, for an optimal application of both rules, it is recommended that the simplification strategy applied to the goal generated by **CUT** starts with the **CS** rule. Otherwise, previous applications of rules such as **EPS**, **CVUL**, or **ICC** (among others) may alter the new equalities and lose any possible match.

Requirements of the command

No fresh variables are introduced in the provided equalities, i.e., the set of variables is contained in the set of variables of the goal's multiclause.

5 Some additional examples

In this section, we present a collection of examples that show how various simplification and induction rules can be used together to improve the theorem proving capabilities of NuITP.

5.1 Multiclause simplification

We begin with a simple example that shows the application of the equality predicate simplification rule (**EPS**).

Consider the following equational theory, which consists of the specification of the natural numbers using Peano notation with the addition, multiplication, and exponentiation operations. It also includes two constructor symbols, namely `[_,_]` and `{_,_}`, of sorts `Pair` and `UPair`, which represent ordered and unordered pairs of numbers, respectively.[24]

```
fmod NAT-ARITH&PAIRS is
  sorts Nat NzNat Zero .
  subsorts Zero NzNat < Nat .

  op 0 : -> Zero [ctor rpo 1] .
  op s : Nat -> NzNat [ctor rpo 2] .

  sorts Pair UPair .

  op [_,_] : Nat Nat -> Pair [ctor rpo 3] .
  op {_,_} : Nat Nat -> UPair [ctor comm rpo 4] .

  vars N M : Nat .

  op _+_ : Nat Nat -> Nat [assoc comm rpo 5 prec 33] .
  eq N + 0 = N .
  eq N + s(M) = s(N + M) .

  op _*_ : Nat Nat -> Nat [assoc comm rpo 6 prec 31] .
  eq N * 0 = 0 .
  eq N * s(0) = N .
  eq N * s(s(M)) = N + (N * s(M)) .

  op _^_ : Nat Nat -> Nat [assoc rpo 7 prec 29] .
  eq N ^ 0 = s(0) .
  eq N ^ s(M) = N * (N ^ M) .
endfm
```

After starting NuITP with the module previously loaded into the Maude system, we first set this module as the active module with the following NuITP command:

```
NuITP> set module NAT-ARITH&PAIRS .

  Module NAT-ARITH&PAIRS is now active.
```

Then, we set the goal[25] we want to prove or, in this case, simplify:

```
NuITP> set goal {X:Nat ^ s(s(0)), Y:Nat} = {s(Y:Nat), 0} \
>             -> [X:Nat + X:Nat ^ s(s(0)), (X:Nat * X:Nat) ] = [s(X:Nat + Y:Nat), X:Nat] .

  Initial goal set.

  Goal Id: 0
  Generated By: init
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
```

[24]Note the `comm` attribute in the declaration of the second operator.

[25]All NuITP commands must be written in a single line, or using the backslash as continuation operator (see Section 6.2 for more info).

```
    None
  Non-Executable Hypotheses:
    None
  Goal:
    {0, s($2:Nat)} = {$2:Nat, $1:Nat ^ s(s(0))}
      -> [s($1:Nat + $2:Nat), $1:Nat] = [$1:Nat + $1:Nat ^ s(s(0)), $1:Nat * $1:Nat]
```

This goal states that if the two unordered pairs to the left of the implication are equal, then the two ordered pairs to the right are also equal.

We are now ready to simplify our goal, which has identifier `0`, by applying the `EPS` rule:

```
NuITP> apply eps to 0 .

  Equality Predicate Simplification (EPS) applied to goal 0.

  Goal Id: 0.1
  Generated By: EPS
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    0 = $2:Nat /\ s($2:Nat) = $1:Nat * $1:Nat
      -> $1:Nat = $1:Nat * $1:Nat /\ s($2:Nat + $1:Nat) = $1:Nat + $1:Nat * $1:Nat
```

The execution of this command ends with the generation of a new goal, result of the simplification of the previous one.

Note that, just by equality predicate simplification, the prover was able to find out that, for the equality in the condition to be true, the variable `Y:Nat` (`$2:Nat`) must be equal to `0` (`0 = $2:Nat`). The rule has simplified a complex clause that used multiplication, addition, power, and ordered and unordered pairs into a much simpler multiclause that only uses addition and multiplication operations.

5.2 Associativity of list concatenation

Consider the following module that specifies the natural numbers in Peano notation, a constructor symbol `_;_` that builds lists of numbers (with `nil` representing the empty list), and a symbol `_@_` for list concatenation.

```
fmod LIST-APPEND is
  sorts Nat List .

  op 0 : -> Nat [ctor rpo 1] .
  op s : Nat -> Nat [ctor rpo 2] .

  op nil : -> List [ctor rpo 3] .
  op _;_ : Nat List -> List [ctor rpo 4] .

  op _@_ : List List -> List [rpo 5] .
  eq nil @ L:List = L:List .
  eq (N:Nat ; L:List) @ Q:List = N:Nat ; (L:List @ Q:List) .
endfm
```

As usual, first we set our module as the active module:

```
NuITP> set module LIST-APPEND .

  Module LIST-APPEND is now active.
```

We want to prove that list concatenation is associative, that is, that the _@_ operator is associative. We first set the following goal as the initial goal:

```
NuITP> set goal (L:List @ P:List) @ Q:List = L:List @ (P:List @ Q:List) .

  Initial goal set.

  Goal Id: 0
  Generated By: init
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $1:List @ ($2:List @ $3:List) = ($1:List @ $2:List) @ $3:List
```

For the application of the **GSI** rule, we need to decide on which variable of the initial goal'a clause we are going to apply the **GSI** induction principle. Let us apply it on variable `L:List`. We also need to think about a suitable generator set for that variable, which is of the `List` sort. For this example, we can use `nil ;; (m:Nat ; R:List)` as our generator set, since any ground constructor term instantiating `L:List` must be either the empty list or a list consisting of a natural number as the head and another list as the tail. Note that the different alternatives in our generator set are separated by using a double semicolon.

We are now ready to apply the **GSI** rule as follows:

```
NuITP> apply gsi to 0 on $1 with nil ;; (m:Nat ; R:List) .

  Generator Set Induction (GSI) applied to goal 0.

  Goal Id: 0.1
  Generated By: GSI
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    nil @ ($2:List @ $3:List) = (nil @ $2:List) @ $3:List

  Goal Id: 0.2
  Generated By: GSI
  Skolem Ops:
    $4.List
    $5.Nat
  Executable Hypotheses:
    ($4 @ $2:List) @ $3:List => $4 @ ($2:List @ $3:List)
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    ($5 ; $4) @ ($2:List @ $3:List) = (($5 ; $4) @ $2:List) @ $3:List
```

The output of the command shows the two subgoals that have been created. We can observe that Goal `0.1` is the result of instantiating the chosen variable `$1:List` in Goal `0` by `nil`. In Goal `0.2` the same variable has been instantiated by the term `($5 ; $4)`, which is itself an instance of the second term in the generator set we provided. Note also that both `$4` and `$5` are so-called *Skolem constants*.

Usually, after applying an induction rule (or even some simplification ones), we want to simplify the newly created goals by applying the **EPS** rule, since there is a good chance the simplification process may succeed in proving those goals, or at least simplifying them. In our example, we can simplify goals `0.1` and `0.2` by applying **EPS** as follows:

```
NuITP> apply eps to 0.1 .

  Equality Predicate Simplification (EPS) applied to goal 0.1.

  Goal 0.1.1 has been proved.

  Unproven goals:

  Goal Id: 0.2
  Generated By: GSI
  Skolem Ops:
    $4.List
    $5.Nat
  Executable Hypotheses:
    ($4 @ $2:List) @ $3:List => $4 @ ($2:List @ $3:List)
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    ($5 ; $4) @ ($2:List @ $3:List) = (($5 ; $4) @ $2:List) @ $3:List

  Total unproven goals: 1

NuITP> apply eps to 0.2 .

  Equality Predicate Simplification (EPS) applied to goal 0.2.

  Goal 0.2.1 has been proved.

  qed
```

By displaying the `qed` acronym (*quod erat demonstrandum*) the prover indicates that the proof has been completed, since both subgoals have been proved and no more goals remain unproved.

Note that, instead of applying the **GSI** rule and then the **EPS** one to each subgoal after setting our initial goal, we could have applied the **GSI!** rule, which automatically simplifies the resulting goals by using the **EPS** rule:

```
NuITP> undo 0 .

  Goal 0 undone. All its children have been deleted.

  Unproven goals:

  Goal Id: 0
  Generated By: init
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $1:List @ ($2:List @ $3:List) = ($1:List @ $2:List) @ $3:List

  Total unproven goals: 1

NuITP> apply gsi! to 0 on $1 with nil ;; (m:Nat ; R:List) .

  Generator Set Induction (GSI) with Equality Predicate Simplification applied to goal 0.

  Goals 0.1.1 and 0.2.1 have been proved.

  qed
```

As we have shown, the **GSI** rule is a powerful induction rule that can help prove certain goals easily. However, its correctness heavily relies on the correctness of the provided generator set, meaning that a faulty or incomplete one that does not cover all possible values for our chosen variable will result in a faulty or incomplete proof.

5.3 Reversing (non-empty) lists

In this example, we will show how to combine rules **EPS** and **GSI** with the narrowing induction rule **NI**.

Consider the following equational theory encoding an associative constructor symbol `__` for non-empty lists of elements, and a predicate `rev` that reverses such lists.

```
fmod REVERSING-LISTS is
  sorts Elt List .
  subsort Elt < List .

  op __ : List List -> List [ctor assoc rpo 1] .

  op rev : List -> List [rpo 2] .
  eq rev(X:Elt) = X:Elt .
  eq rev(X:Elt L:List) = rev(L:List) X:Elt .
endfm
```

We begin by setting our functional module as the active module:

```
NuITP> set module REVERSING-LISTS .

  Module REVERSING-LISTS is now active.
```

We want to prove that the reverse of a list of the form `Q:List Y:Elt` is equal to the element `Y:Elt` concatenated with the reverse of the list `Q:List`. For that we set our goal as follows:

```
NuITP> set goal rev(Q:List Y:Elt) = Y:Elt rev(Q:List) .

  Initial goal set.

  Goal Id: 0
  Generated By: init
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    rev($1:List $2:Elt) = $2:Elt rev($1:List)
```

We could try using the **GSI** rule, but instead we use narrowing induction by applying the **NI** rule on the subterm `rev($1:List $2:Elt)` of the clause:

```
NuITP> apply ni to 0 on rev($1:List $2:Elt) .

  Narrowing Induction (NI) applied to goal 0.

  Goal Id: 0.1
  Generated By: NI
  Skolem Ops:
    $3.Elt
    $4.Elt
    $5.List
  Executable Hypotheses:
    rev($5 $4) => $4 rev($5)
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $4 rev($3 $5) = rev($5 $4) $3

  Goal Id: 0.2
  Generated By: NI
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $4:Elt rev($3:Elt) = rev($4:Elt) $3:Elt
```

This command basically *narrows* the term (i.e., it symbolically evaluates the term with the equations defining the `rev` function), yielding the shown two goals. Goal `0.1` has now a ground clause, where fresh variables `$3:Elt`, `$4:Elt`, and `$5:List`, which were generated by the narrowing algorithm, have been converted into *Skolem* constants of their respective sorts. Moreover, an executable (i.e., oriented, note the symbol => in the equality) hypothesis has also been generated. We can now prove Goal `0.1` by applying the **EPS** rule:

```
NuITP> apply eps to 0.1 .

  Equality Predicate Simplification (EPS) applied to goal 0.1.

  Goal 0.1.1 has been proved.

  Unproven goals:

  Goal Id: 0.2
  Generated By: NI
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $4:Elt rev($3:Elt) = rev($4:Elt) $3:Elt

  Total unproven goals: 1
```

Goal `0.2` is not ground and has not generated any hypotheses. However, it can be trivially proved by using the very same equations of the original theory, which state that the reverse of an element is the element itself (`rev(X:Elt) = X:Elt`). Hence, we also apply the **EPS** rule on this goal:

```
NuITP> apply eps to 0.2 .

  Equality Predicate Simplification (EPS) applied to goal 0.2.

  Goal 0.2.1 has been proved.

  qed
```

As usual, we could have shorten our proof by using the **NI!** rule after setting the initial goal, which would apply narrowing induction followed by equality predicate simplification:

```
NuITP> undo 0 .

  Goal 0 undone. All its children have been deleted.

  Unproven goals:

  Goal Id: 0
  Generated By: init
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    rev($1:List $2:Elt) = $2:Elt rev($1:List)

  Total unproven goals: 1

NuITP> apply ni! to 0 on rev($1:List $2:Elt) .

  Narrowing Induction (NI) with Equality Predicate Simplification applied to goal 0.

  Goals 0.1.1 and 0.2.1 have been proved.

  qed
```

In the following, we will use the extended, `!` versions of NuITP's induction rules to shorten the presentation when there is no need to show the details of the intermediate steps, since we are most likely interested in simplifying our new goals with the **EPS** rule before trying another rule applications.

5.4 Using lemmas

Sometimes, when trying to prove a goal, we need auxiliary lemmas that either have been previously proved or will be proved together with our initial goal. In this example, we will show how we can add such lemmas by using the **LE** rule. Additionally, we will show how to use the **CAS** and **EQ** rules, which will help proving our initial goal.

Let us consider the usual equational theory defining natural numbers in Peano notation with the addition and multiplication operations:

```
fmod PEANO+RAxR is
  sorts Nat NzNat Zero .
  subsorts Zero NzNat < Nat .

  op 0 : -> Zero [ctor rpo 1] .
  op s_ : Nat -> NzNat [ctor rpo 2] .

  op _+_ : Nat Nat -> Nat [assoc rpo 3] .
  eq N:Nat + 0 = N:Nat .
  eq N:Nat + s M:Nat = s(N:Nat + M:Nat) .

  op _*_ : Nat Nat -> Nat [assoc comm rpo 4] .
  eq N:Nat * 0 = 0 .
  eq N:Nat * s 0 = N:Nat .
  eq N:Nat * s s M:Nat = N:Nat + (N:Nat * s M:Nat) .
endfm
```

Note that, in the above specification, addition is declared associative but not commutative, which is the property that we will need to add as a lemma in our proof.

As usual, we start setting our module as the active module for the session and declaring the generator sets for the sorts of the module:

```
NuITP> set module PEANO+RAxR .

  Module PEANO+RAxR is now active.

NuITP> genset GEN-NAT for Nat is 0 ;; s N:Nat .

  Generator set GEN-NAT for sort Nat added.

  GEN-NAT (default):
    0
    s N:Nat

NuITP> genset GEN-NZNAT for NzNat is s N:Nat .

  Generator set GEN-NZNAT for sort NzNat added.

  GEN-NZNAT (default):
    s N:Nat

NuITP> genset GEN-ZERO for Zero is 0 .

  Generator set GEN-ZERO for sort Zero added.

  GEN-ZERO (default):
    0
```

Then, we declare this simple goal:

```
NuITP> set goal X:Nat * X:Nat = s 0 -> s 0 = X:Nat .
```

```
  Initial goal set.

  Goal Id: 0
  Generated By: init
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    s 0 = $1:Nat * $1:Nat -> $1:Nat = s 0
```

Now, we can apply the case rule **CAS** on the `X:Nat` variable of our initial goal by specifying a suitable generator set `0 ;; s(0) ;; s(s(Z:Nat))` for natural numbers in Peano notation:

```
NuITP> apply cas to 0 on $1:Nat with 0 ;; s 0 ;; s s Z:Nat .

  Case (CAS) applied to goal 0.

  Goal Id: 0.1
  Generated By: CAS
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    s 0 = 0 * 0 -> 0 = s 0

  Goal Id: 0.2
  Generated By: CAS
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    s 0 = s 0 * s 0 -> s 0 = s 0

  Goal Id: 0.3
  Generated By: CAS
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    s 0 = s s $2:Nat * s s $2:Nat -> s 0 = s s $2:Nat
```

Goals `0.1` and `0.2` can be easily proved by applying the **EPS** simplification rule:

```
NuITP> apply eps to 0.1 .

  Equality Predicate Simplification (EPS) applied to goal 0.1.

  Goal 0.1.1 has been proved.

  Unproven goals:

  Goal Id: 0.2
  Generated By: CAS
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    s 0 = s 0 * s 0 -> s 0 = s 0

  Goal Id: 0.3
  Generated By: CAS
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    s 0 = s s $2:Nat * s s $2:Nat -> s 0 = s s $2:Nat

  Total unproven goals: 2

NuITP> apply eps to 0.2 .

  Equality Predicate Simplification (EPS) applied to goal 0.2.

  Goal 0.2.1 has been proved.

  Unproven goals:

  Goal Id: 0.3
  Generated By: CAS
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    s 0 = s s $2:Nat * s s $2:Nat -> s 0 = s s $2:Nat

  Total unproven goals: 1
```

However, simplifying Goal `0.3` with **EPS** will still not prove it:

```
NuITP> apply eps to 0.3 .

  Equality Predicate Simplification (EPS) applied to goal 0.3.

  Goal Id: 0.3.1
  Generated By: EPS
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    s 0 = s (s s $2:Nat + $2:Nat) + s $2:Nat * s $2:Nat -> false
```

Needless to say that these steps could have been avoided by the use of the `!` strategy.

```
NuITP> undo 0 .

  Goal 0 undone. All its children have been deleted.

  Unproven goals:

  Goal Id: 0
  Generated By: init
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    s 0 = $1:Nat * $1:Nat -> $1:Nat = s 0

  Total unproven goals: 1

NuITP> apply cas! to 0 on $1:Nat with 0 ;; s 0 ;; s s Z:Nat .

  Case (CAS) with Equality Predicate Simplification applied to goal 0.

  Goals 0.1.1 and 0.2.1 have been proved.

  Goal Id: 0.3.1
  Generated By: EPS
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    s 0 = s (s s $2:Nat + $2:Nat) + s $2:Nat * s $2:Nat -> false
```

At this point we can *enrich* our theory by means of a new lemma that will help us in the proving process. Specifically, commutativity of the addition operator will be helpful. We can do this by applying the **LE** rule in the following way:

```
NuITP> apply le to 0.3.1 with N:Nat + M:Nat = M:Nat + N:Nat .

  Lemma Enrichment (LE) applied to goal 0.3.1.

  Goal Id: 0.3.1.1
  Generated By: LE
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $4:Nat + $3:Nat = $3:Nat + $4:Nat

  Goal Id: 0.3.1.2
  Generated By: LE
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    $4:Nat + $3:Nat = $3:Nat + $4:Nat
  Goal:
    s 0 = s (s s $2:Nat + $2:Nat) + s $2:Nat * s $2:Nat -> false
```

Note that the application of the **LE** rule to Goal `0.3.1` produces two new subgoals, namely, `0.3.1.1` and `0.3.1.2`. The first one is actually the lemma we have introduced, which needs to be proved, and the second one adds this new lemma to the theory of the original goal as a hypothesis. Unfortunately, since this new hypothesis is intrinsically non-terminating, it remains as a non-executable hypothesis and we cannot use it as a rewrite rule to help proving our goal. We can, however, use the hypothesis in a controlled way using the **EQ** rule or its **EQ!** extension with **EPS** simplification. To apply it, we just need to specify how we want to orient it. In this case, we can choose either `N:Nat + M:Nat => M:Nat + N:Nat` or `M:Nat + N:Nat => N:Nat + M:Nat`.

```
NuITP> apply eq! to 0.3.1.2 with ($4:Nat + $3:Nat) => $3:Nat + $4:Nat .

  Multiple matches found for EQ application. Please select one.

  Matching #0
    s 0 = s (s s $2:Nat + $2:Nat) + s $2:Nat * s $2:Nat -> false
  with substitution $3:Nat <- s $2:Nat * s $2:Nat ; $4:Nat <- s (s s $2:Nat + $2:Nat)

  Matching #1
    s 0 = s s s $2:Nat + $2:Nat + s $2:Nat * s $2:Nat -> false
  with substitution $3:Nat <- $2:Nat ; $4:Nat <- s s $2:Nat

  Select valid matching number or q(uit): 1

  Equality (EQ) with Equality Predicate Simplification applied to goal 0.3.1.2.

  Goal 0.3.1.2.1.1 has been proved.

  Unproven goals:

  Goal Id: 0.3.1.1
  Generated By: LE
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $4:Nat + $3:Nat = $3:Nat + $4:Nat

  Total unproven goals: 1
```

Note that, in this case, the hypothesis can be applied in two different ways: either with substitution {`$3:Nat` ← `s $2:Nat * s $2:Nat` ; `$4:Nat` ← `s (s s $2:Nat + $2:Nat)`} or with substitution {`$3:Nat` ← `$2:Nat` ; `$4:Nat` ← `s s $2:Nat`}. The tool shows all the possibilities and we must chose which one we are interested in. In this case, option `1` was chosen, which closes the proof of the goal and leaves as with one single pending goal.

Alternatively, we can provide a substitution to the `EQ` command, automating the application of the hypothesis in the desired way.

```
NuITP> undo 0.3.1.2 .

  Goal 0.3.1.2 undone. All its children have been deleted.

  Unproven goals:

  Goal Id: 0.3.1.1
  Generated By: LE
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $3:Nat + $4:Nat = $4:Nat + $3:Nat

  Goal Id: 0.3.1.2
  Generated By: LE
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    $3:Nat + $4:Nat = $4:Nat + $3:Nat
  Goal:
    s 0 = s (s s $2:Nat + $2:Nat) + s $2:Nat * s $2:Nat -> false

  Total unproven goals: 2

NuITP> apply eq! to 0.3.1.2 with $4:Nat + $3:Nat => $3:Nat + $4:Nat \
>                             sub $3:Nat <- $2:Nat ; $4:Nat <- s s $2:Nat .

  Equality (EQ) with Equality Predicate Simplification applied to goal 0.3.1.2.

  Goal 0.3.1.2.1.1 has been proved.
  Unproven goals:

  Goal Id: 0.3.1.1
  Generated By: LE
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $3:Nat + $4:Nat = $4:Nat + $3:Nat

  Total unproven goals: 1
```

At this point, the only remaining goal is the lemma we introduced with the **LE** rule, which needs to be proved[26] in the current session, but, for this example, let us assume we already did it in a previous session and conclude that we have successfully proved our initial goal.

5.5 Multiplicative cancellation

In this example, we will use a variety of rules all combined to prove our goal, namely **CVUL**, **CS**, **GSI**, **VA**, **CAS** in both variables and *Skolem* constants, and, of course, **EPS**, since we will use the extended versions of the rules that apply **EPS** automatically.

Consider the following equational theory defining Presburger arithmetic for the natural numbers (`_+_` and `_>_`), which we have extended with the `_*_` multiplication operator. Note than both addition and multiplication are declared with associative and commutative axioms. Note also that the subset of equations that define the `_>_` and `_+_` symbols have the `variant` attribute, since they have the finite variant property (thus making Presburger arithmetic decidable by the so-called variant satisfiability procedure), but not the equations defining the `_*_`, which are not FVP.

```
fmod PRESBURGER&MULT is
  sorts Bool Zero NzNat Nat .
  subsorts Zero NzNat < Nat .

  op true : -> Bool [ctor rpo 1] .
  op false : -> Bool [ctor rpo 2] .
  op 0 : -> Zero [ctor rpo 3] .
  op 1 : -> NzNat [ctor rpo 4] .

  op _>_ : Nat Nat -> Bool [rpo 5] .
  eq (X:Nat + X':NzNat) > X:Nat = true [variant] .
  eq X:Nat > (X:Nat + Y:Nat) = false [variant] .

  op _+_ : Nat Nat -> Nat [assoc comm rpo 6] .
  op _+_ : Nat NzNat -> NzNat [ditto] .
  op _+_ : NzNat Nat -> NzNat [ditto] .
  op _+_ : NzNat NzNat -> NzNat [ctor ditto] .
  eq X:Nat + 0 = X:Nat [variant] .

  op _*_ : Nat Nat -> Nat [assoc comm rpo 7] .
  op _*_ : NzNat NzNat -> NzNat [ditto] .
  eq X:Nat * 0 = 0 .
  eq X:Nat * 1 = X:Nat .
  eq X:Nat * (Y:Nat + Z:Nat) = (X:Nat * Y:Nat) + (X:Nat * Z:Nat) .
endfm
```

[26]Two alternative proofs of the commutativity of addition can be found in Section 2.4.

```
NuITP> set module PRESBURGER&MULT .

  Module PRESBURGER&MULT is now active.
```

We want to prove the cancellation law for natural numbers multiplication so, after setting as active our module above, we set the following initial goal:

```
NuITP> set goal X:Nat * Z':NzNat = Y:Nat * Z':NzNat -> X:Nat = Y:Nat .

  Initial goal set.

  Goal Id: 0
  Generated By: init
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $1:Nat * $3:NzNat = $2:Nat * $3:NzNat -> $1:Nat = $2:Nat
```

We start proving our goal by first applying our well known **GSI!** rule, which will apply **GSI** followed by **EPS** in each of the generated goals:

```
NuITP> apply gsi! to 0 on $1 with 0 ;; 1 ;; 1 + X:NzNat .

  Generator Set Induction (GSI) with Equality Predicate Simplification applied to goal 0.

  Goal Id: 0.1.1
  Generated By: EPS
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    0 = $3:NzNat * $2:Nat -> 0 = $2:Nat

  Goal Id: 0.2.1
  Generated By: EPS
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $3:NzNat = $3:NzNat * $2:Nat -> 1 = $2:Nat

  Goal Id: 0.3.1
  Generated By: EPS
  Skolem Ops:
    $4.NzNat
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    $3:NzNat = $2:Nat * $3:NzNat -> 1 => $2:Nat
    $4 * $3:NzNat = $2:Nat * $3:NzNat -> $4 => $2:Nat
  Goal:
    $3:NzNat * $2:Nat = $3:NzNat + $4 * $3:NzNat -> $2:Nat = $4 + 1
```

To prove Goal `0.1.1`, we apply **CAS!** on variable `$2:Nat` as follows:

```
NuITP> apply cas! to 0.1.1 on $2 with 0 ;; Y:NzNat .

  Case (CAS) with Equality Predicate Simplification applied to goal 0.1.1.

  Goal 0.1.1.2.1 has been proved.

  Goal Id: 0.1.1.1
  Generated By: CAS
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    0 = $3:NzNat * $4:NzNat -> 0 = $4:NzNat
```

Then, we can use variable abstraction (**VA**), which will abstract the subterm of the clause that we provide as argument (i.e., `$3:NzNat * $4:NzNat`) into a new, fresh variable:

```
NuITP> apply va! to 0.1.1.1 on $3:NzNat * $4:NzNat .

  Variable Abstraction (VA) with Equality Predicate Simplification applied to goal 0.1.1.1.

  Goal Id: 0.1.1.1.1
  Generated By: VA
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    0 = $5:NzNat /\ $5:NzNat = $3:NzNat * $4:NzNat -> 0 = $4:NzNat
```

This command leaves the clause in Goal `0.1.1.1.1` ready to be simplified by means of the **CVUL** simplification rule,[27] since the equality `0 = $5:NzNat` will never be true, which will falsify the condition proving the premise:

```
NuITP> apply cvul to 0.1.1.1.1 .

  Constructor Variant Unification Left (CVUL) applied to goal 0.1.1.1.1.
```

[27]Note that the new abstraction variable uses the goal identifier to avoid undesirable clashes.

```
  Goal 0.1.1.1.1.1 has been proved.

  Unproven goals:

  Goal Id: 0.2.1
  Generated By: EPS
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $3:NzNat = $3:NzNat * $2:Nat -> 1 = $2:Nat

  Goal Id: 0.3.1
  Generated By: EPS
  Skolem Ops:
    $4.NzNat
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    $3:NzNat = $2:Nat * $3:NzNat -> 1 => $2:Nat
    $4 * $3:NzNat = $2:Nat * $3:NzNat -> $4 => $2:Nat
  Goal:
    $3:NzNat * $2:Nat = $3:NzNat + $4 * $3:NzNat -> $2:Nat = $4 + 1

  Total unproven goals: 2
```

Now we try to prove Goal `0.2.1` in a very similar way by first applying **GSI!**:

```
NuITP> apply gsi! to 0.2.1 on $2 with 0 ;; Y:NzNat .

  Generator Set Induction (GSI) with Equality Predicate Simplification applied to goal 0.2.1.

  Goal Id: 0.2.1.1
  Generated By: GSI
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $3:NzNat = $3:NzNat * $4:NzNat -> 1 = $4:NzNat

  Goal Id: 0.2.1.2.1
  Generated By: EPS
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    0 = $3:NzNat -> false
```

and then **CVUL** to Goal `0.2.1.2.1`:

```
NuITP> apply cvul to 0.2.1.2.1 .

  Constructor Variant Unification Left (CVUL) applied to goal 0.2.1.2.1.

  Goal 0.2.1.2.1.1 has been proved.

  Unproven goals:

  Goal Id: 0.3.1
  Generated By: EPS
  Skolem Ops:
    $4.NzNat
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    $3:NzNat = $2:Nat * $3:NzNat -> 1 => $2:Nat
    $4 * $3:NzNat = $2:Nat * $3:NzNat -> $4 => $2:Nat
  Goal:
    $3:NzNat * $2:Nat = $3:NzNat + $4 * $3:NzNat -> $2:Nat = $4 + 1

  Goal Id: 0.2.1.1
  Generated By: GSI
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $3:NzNat = $3:NzNat * $4:NzNat -> 1 = $4:NzNat

  Total unproven goals: 2
```

and **CAS!** to Goal `0.2.1.1` but, this time, on variable `$4:NzNat`:

```
NuITP> apply cas! to 0.2.1.1 on $4 with 1 ;; 1 + Y:NzNat .

  Case (CAS) with Equality Predicate Simplification applied to goal 0.2.1.1.

  Goals 0.2.1.1.1.1 and 0.2.1.1.2.1 have been proved.

  Unproven goals:

  Goal Id: 0.3.1
  Generated By: EPS
  Skolem Ops:
    $4.NzNat
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    $3:NzNat = $2:Nat * $3:NzNat -> 1 => $2:Nat
    $4 * $3:NzNat = $2:Nat * $3:NzNat -> $4 => $2:Nat
  Goal:
    $3:NzNat * $2:Nat = $3:NzNat + $4 * $3:NzNat -> $2:Nat = $4 + 1

  Total unproven goals: 1
```

Finally, we start simplifying our remaining Goal `0.3.1` by first applying **CAS!**:

```
NuITP> apply cas! to 0.3.1 on $2 with 0 ;; 1 ;; 1 + Y:NzNat .

  Case (CAS) with Equality Predicate Simplification applied to goal 0.3.1.

  Goals 0.3.1.1.1 and 0.3.1.2.1 have been proved.

  Goal Id: 0.3.1.3.1
  Generated By: EPS
  Skolem Ops:
    $4.NzNat
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    $3:NzNat = $2:Nat * $3:NzNat -> 1 => $2:Nat
    $4 * $3:NzNat = $2:Nat * $3:NzNat -> $4 => $2:Nat
  Goal:
    $4 * $3:NzNat = $3:NzNat * $5:NzNat -> $4 = $5:NzNat
```

and then **CS**, as one of the non-executable hypotheses we have computed entirely subsumes the clause of our goal:

```
NuITP> apply cs to 0.3.1.3.1 .

  Clause Subsumption (CS) applied to goal 0.3.1.3.1.

  Goal 0.3.1.3.1.1 has been proved.

  qed
```

Note that, in this example, we have used up to three different generator sets depending on our strategy to prove a goal and the sort of the variable (or *Skolem* constant) on which the rule was going to be applied. Choosing the right generator set in each step can help us to greatly simplify our proofs.

5.6 Reversing palindromes

Consider the following conditional equational theory, which extends the theory of Section 5.3 with a new defined predicate `pal` that, given a non-empty list, evaluates to true or false depending on whether the provided list is a palindrome or not. We also add an auxiliary predicate `_=e=_` that evaluates to true if two given lists are equal, and to false otherwise:

```
load NuITP-lib

set include BOOL off .

fmod NAT-PRESS is
  pr NuITP-BOOL .

  sorts Nat NzNat .
  subsort NzNat < Nat .
  op 0 : -> Nat [ctor rpo 11] .
  op 1 : -> NzNat [ctor rpo 12] .

  op _+_ : Nat Nat -> Nat [prec 33 assoc comm id: 0 rpo 15] .
  op _+_ : Nat NzNat -> NzNat [ditto] .
  op _+_ : NzNat NzNat -> NzNat [ctor ditto] .

  vars N M : Nat .   var P : NzNat .

  op _=e=_ : Nat Nat -> Bool [comm rpo 16] .
  eq [NP-5] : N =e= N = True [variant] .
  eq [NP-6] : N + P =e= N = False [variant] .
endfm

fmod REVERSING-PALINDROMES is
  pr NAT-PRESS .

  sort List .
  subsort Nat < List .
  op __ : List List -> List [ ctor assoc rpo 4 ] .

  op _=e=_ : List List -> Bool [ ditto ] .
  ceq N:Nat L:List =e= M:Nat Q:List = L:List =e= Q:List if N:Nat =e= M:Nat = True .
  eq N:Nat L:List =e= Q:List = False .

  op rev : List -> List [ rpo 3 ] .
  eq rev(N:Nat) = N:Nat .
  eq rev(N:Nat L:List) = rev(L:List) N:Nat .

  op pal : List -> Bool [ rpo 20 ] .
  eq pal(N:Nat) = True .
  eq pal(N:Nat N:Nat) = True .
  eq pal(N:Nat L:List N:Nat) = pal(L:List) .
  ceq pal(N:Nat M:Nat) = False if (N:Nat =e= M:Nat) = False .
  ceq pal(N:Nat L:List M:Nat) = False if (N:Nat =e= M:Nat) = False .
endfm
```

```
NuITP> set module REVERSING-PALINDROMES .

  Module REVERSING-PALINDROMES is now active.
```

For this example, we want to prove the straightforward statement that says that, if a list is a palindrome, then the reverse of that list is the same list:

```
NuITP> set goal pal(L:List) = True -> rev(L:List) = L:List .

  Initial goal set.

  Goal Id: 0
  Generated By: init
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    True = pal($1:List) -> $1:List = rev($1:List)
```

First, we introduce an auxiliary lemma `rev(L:List N:Nat) = N:Nat rev(L:List)` in our current proof, which we already proved to be true in the reversing lists example of Section 5.3:

```
NuITP> apply le! to 0 with rev(L:List N:Nat) = N:Nat rev(L:List) .

  Lemma Enrichment (LE) with Equality Predicate Simplification applied to goal 0.

  Goal Id: 0.1
  Generated By: LE
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    rev($2:List $3:Nat) = $3:Nat rev($2:List)

  Goal Id: 0.2
  Generated By: LE
  Skolem Ops:
    None
  Executable Hypotheses:
    rev($2:List $3:Nat) => $3:Nat rev($2:List)
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    True = pal($1:List) -> $1:List = rev($1:List)
```

This command generates two goals: goal `0.1`, which is actually the lemma we introduced, and goal `0.2`, where the lemma is added to the original goal as an executable hypothesis, which we now try to prove by first applying narrowing induction **NI!** on the subterm `pal(L:List)` of its clause:

```
NuITP> apply ni! to 0.2 on pal($1:List) .

  Narrowing Induction (NI) with Equality Predicate Simplification applied to goal 0.2.

  Goals 0.2.1.1, 0.2.2.1, 0.2.3.1 and 0.2.5.1 have been proved.

  Goal Id: 0.2.4.1
  Generated By: EPS
  Skolem Ops:
    $4.Nat
    $5.List
  Executable Hypotheses:
    rev($2:List $3:Nat) => $3:Nat rev($2:List)
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    True = pal($5) -> rev($5) => $5
  Goal:
    True = pal($5) -> $5 = rev($5)
```

Then, applying clause subsumption (**CS**) to the remaining goal.

```
NuITP> apply cs to 0.2.4.1 .

  Clause Subsumption (CS) applied to goal 0.2.4.1.

  Goal 0.2.4.1.1 has been proved.

  Unproven goals:

  Goal Id: 0.1
  Generated By: LE
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Quasi-Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    rev($2:List $3:Nat) = $3:Nat rev($2:List)

  Total unproven goals: 1
```

Note that **CS** will take advantage of the new non-executable hypothesis we computed, as it subsumes (actually, it is equal to) the clause we want to prove in our goal.

Now the only goal that remains unproved is the lemma we introduced, which was proved in a previous session, so we can prove it exactly as before to conclude the proof of our initial goal.

5.7 Maude module and proof script for Example 2

Consider module introduced in Example 2 in Section 4.5.2.

```
load NuITP-lib

set include BOOL off .

fmod NAT-GT is
  pr NuITP-BOOL .
  sorts NzNat Nat .
  subsort NzNat < Nat .
  op 0 : -> Nat [ctor rpo 13] .
  op 1 : -> NzNat [ctor rpo 14] .
  op _+_ : NzNat NzNat -> NzNat [ctor assoc comm id: 0 rpo 15] .
  op _+_ : Nat Nat -> Nat [ditto] .
  op _>_ : Nat Nat -> Bool [rpo 16] .
  op _-_ : Nat Nat -> Nat [rpo 17] .
  vars N M : Nat .
  eq N + M + 1 > N = True [variant] .
  eq N > N + M = False [variant] .
  eq (M + N) - N = M [variant] .
  eq M - (N + M) = 0 [variant] .
endfm
```

The proof, as discussed in Section 4.5.2, can be executed as follows.

```
NuITP> set module NAT-GT .

  Module NAT-GT is now active.

NuITP> set goal (X:Nat > Y:Nat = True) -> (X:Nat - Y:Nat > 0 = True) .
```

```
    Initial goal set.

    Goal Id: 0
    Generated By: init
    Skolem Ops:
      None
    Executable Hypotheses:
      None
    Quasi-Executable Hypotheses:
      None
    Non-Executable Hypotheses:
      None
    Goal:
      True = $1:Nat > $2:Nat -> True = ($1:Nat - $2:Nat) > 0

  NuITP> apply cvul! to 0 .

    Constructor Variant Unification Left (CVUL) with Equality Predicate Simplification applied to goal 0.

    Goal 0.1.1 has been proved.

    qed
```

6 Troubleshooting

In the following, we describe some requirements and common problems that can arise while running NuITP and how to avoid them.

6.1 Multi-line inputs

The backslash continuation character can be used to write multi-line inputs.

```
  NuITP> set goal (Z:Nat * (X:Nat + Y:Nat) = (Z:Nat * X:Nat) + (Z:Nat * Y:Nat)) \
         /\ (((X:Nat + Y:Nat) * Z:Nat) = (X:Nat * Z:Nat) + (Y:Nat * Z:Nat)) .

    Initial goal set.

    Goal Id: 0
    Generated By: init
    Skolem Ops:
      None
    Executable Hypotheses:
      None
    Quasi-Executable Hypotheses:
      None
    Non-Executable Hypotheses:
      None
    Goal:
      ($3:Nat * ($1:Nat + $2:Nat) = $3:Nat * $1:Nat + $3:Nat * $2:Nat) /\ $3:Nat *
      ($1:Nat + $2:Nat) = $3:Nat * $1:Nat + $3:Nat * $2:Nat
```

6.2 Common parsing problems

Any NuITP command is expected in one single line. Each time a new line is added, the tool attempts parsing the input. Therefore:

- An empty line will produce an error message:

```
NuITP>

  Error parsing command.
```

- A command with a carriage return will be considered as two separated commands, resulting in two wrong inputs:

```
NuITP> set goal (Z:Nat * (X:Nat + Y:Nat) = (Z:Nat * X:Nat) + (Z:Nat * Y:Nat))

  Error parsing command.

NuITP>       /\ (((X:Nat + Y:Nat) * Z:Nat) = (X:Nat * Z:Nat) + (Y:Nat * Z:Nat)) .

  Error parsing command.
```

  See Section 6.1 for information on the use of the backslash to enter multi-line inputs.

- The backslash character does not work in script files.

6.3 Enabling I/O operations on files

NuITP allows users to load (resp. save) scripts from (resp. to) files, as well as generate LaTeX documents with a summary of the current session, and serialize it to a file in order to resume a NuITP session from the point it was saved. All of this is done by taking advantage of the latest Maude I/O capabilities. Since accessing the file system represents a potential security threat, Maude has I/O operations disabled by default. Therefore, in order to be able to use these NuITP features successfully, the Maude interpreted needs to be initialized with the -allow-files or -trust flags (see Chapter 9.2 of [2]) as follows:

```
$maude -allow-files NuITP.maude
```

or

```
$maude -trust NuITP.maude
```

Failing to initialize Maude with the `-allow-files` flag will result in the inability to use the `load`, saving commands (`save`, `snap`, `tex`, and `html`), but should not affect the remaining features of NuITP.

6.4 Writing and running files

As pointed out above, NuITP relies on Maude's I/O capabilities to access the file system, which in turn wraps the C *stdio* library (see [2, Chapter 9]). Access to files will therefore be limited by the permissions the current user has been granted by the operative system. Proper failure messages will be forwarding for NuITP to show them in case files are not accessible.

Beware also that, as of its current alpha 36 version, NuITP saves scripts, proof reports, and running sessions without asking for confirmation on the provided file name, so it will overwrite the specified file if the file already exists. It is the user's responsibility to provide a *safe*[28] filename that will not result in a potential risk.

7 References

[1] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[2] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, R. Rubio, and C. Talcott. Maude Manual (Version 3.5.1). Technical report, SRI International Computer Science Laboratory, 2024. Available at: http://maude.cs.illinois.edu.

[28]Remember that NuITP automatically adds the proper extension to the filename if it is omitted.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[4] H. Comon-Lundh and S. Delaune. The Finite Variant Property: How to Get Rid of Some Algebraic Properties. In *16th International Conference on Rewriting Techniques and Applications (RTA)*, volume 3467 of *Lecture Notes in Computer Science*, pages 294–307. Springer, 2005.

[5] F. Durán, S. Escobar, J. Meseguer, and J. Sapiña. NuITP: An Inductive Theorem Prover for Equational Program Verification. In *Proceedings of the 26th International Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 6:1–6:11. ACM Press, 2024.

[6] F. Durán, S. Escobar, J. Meseguer, and J. Sapiña. NuITP: An Inductive Theorem Prover for Equational Program Verification. In *Actas de las XXIV Jornadas de Programación y Lenguajes (PROLE)*. Sistedes, 2025.

[7] S. Escobar, R. Sasse, and J. Meseguer. Folding Variant Narrowing and Optimal Variant Termination. *The Journal of Logic and Algebraic Programming*, 81(7–8):898–928, 2012.

[8] R. Gutiérrez, J. Meseguer, and C. Rocha. Order-Sorted Equality Enrichments Modulo Axioms. *Science of Computer Programming*, 99:235–261, 2015.

[9] R. Gutiérrez, J. Meseguer, and S. Skeirik. The maude termination assistant. In *Preproceedings of International Workshop on Rewriting Logic and its Applications (WRLA)*, 2018.

[10] J. Hendrix, J. Meseguer, and H. Ohsaki. A Sufficient Completeness Checker for Linear Order-Sorted Specifications Modulo Axioms. In *3rd International Joint Conference on Automated Reasoning (IJCAR)*, volume 4130 of *Lecture Notes in Computer Science*, pages 151–155. Springer, 2006.

[11] S. Lucas and J. Meseguer. Normal forms and normal theories in conditional rewriting. *J. Log. Algebr. Meth. Program.*, 85(1):67–97, 2016.

[12] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[13] J. Meseguer. Order-Sorted Rewriting and Congruence Closure. In *Proceedings of the 19th International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, volume 9634 of *Lecture Notes in Computer Science*, pages 493–509. Springer, 2016.

[14] J. Meseguer. Variant-Based Satisfiability in Initial Algebras. *Science of Computer Programming*, 154:3–41, 2018.

[15] J. Meseguer. Equivalence, and property internalization and preservation for equational programs. In *Rewriting Logic and Its Applications*, pages 62–83, Cham, 2024. Springer.

[16] J. Meseguer. Inductive Reasoning with Equality Predicates, Contextual Rewriting and Variant-Based Simplification. *Journal of Logical and Algebraic Methods in Programming*, 144:101036, 2025.

[17] S. Skeirik and J. Meseguer. Metalevel Algorithms for Variant Satisfiability. *Journal of Logical and Algebraic Methods in Programming*, 96:81–110, 2018.

[18] S. Skeirik and J. Meseguer. Metalevel Algorithms for Variant Satisfiability. *Journal of Logical and Algebraic Methods in Programming*, 96:81–110, 2018.