NuITP

alpha 28

An Inductive Theorem Prover for Maude Equational Theories

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

April 2024

Contents

1 Getting started

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.

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 (i.e., version 3.4). Otherwise, the requirements are those of the Maude's interpreter itself.

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

1.1 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 [7], 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.

The 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 4.5 and [2, Chapter 14] for more info). 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.[1]

On the FVP subtheory:

- Equations in E_1 must be unconditional, and must be declared with the `variant` attribute[2].
- 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 $E \setminus E_1$ can be conditional, but their conditions should not have any extra variables not present in their lefthand side. Furthermore, they should *not* use any built-in features such as the `==` equality predicate or the `owise` attribute.

[1] 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:

```
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.

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

1.2 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

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

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:

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

$$\ldots$$

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

A multiclause with no condition, i.e.,

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

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

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

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 will be written in their typed form, that is, a variable N of sort *Nat* will be written as `N:Nat`. The parser may also need some additional parentheses to get things right. 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.

1.3 RPO termination order specified with the `metadata` 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 the 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[3] 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 different modulo B can always be compared under the $\prec$ order, which is very useful for some of the NuITP inference rules. To achieve these termination properties, NuITP requires that the user specifies a suitable RPO

[3] Such axioms should not include identity axioms. This is ensured by the 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.

order modulo axioms that is total on function symbols by *ordering* the signature of the input theory by means of a *tagging* of each of its operators with a natural number using the `metadata` 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 metadata "1" ] .
  op s_ : Nat -> NzNat [ ctor metadata "2" ] .

  op _+_ : Nat Nat -> Nat [ assoc comm metadata "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 metadata 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.

1.4 Running NuITP

NuITP runs with the latest version of the Maude System (version 3.4),[4] which can be downloaded from the Maude's website at `http://maude.cs.illinois.edu`.

The current version of NuITP is distributed as a Maude file, named `NuITP.maude`. To run the tool simply load the `NuITP.maude` file by providing it as an argument when starting the Maude System or by loading it manually by means of Maude's `load` command. Once loaded, the tool will automatically start. To be able to read from and write into files, Maude requires to be run with the `allow-files` or `trust` flags on (see the Maude manual [2, Chapter 9]). After loading the `NuITP.maude` file you should see the tool's prompt:

[4]Note that NuITP makes use of some functions declared in the `file.maude`, 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.

```
$ maude -allow-files NuITP.maude

          \||||||||||||||||||/
        --- Welcome to Maude ---
          /||||||||||||||||||\
 Maude 3.3.1 built: Apr 13 2023 16:10:31
  Copyright 1997-2023 SRI International
        Thu Jul 6 09:33:00 2023
==========================================
erewrite in NuITP : init .


    ==================================
              NuITP (alpha 21)
           Inductive Theorem Prover
       for Maude Equational Theories
    ==================================
            Copyright 2021-2023
    Universitat Politècnica de València

NuITP>
```

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 you must learn are:

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

`help` to get a basic help on the syntax on available commands.[5]

Before explaining the NuITP commands in detail, we will present some basic information on the tool and how to interact with it. NuITP is written in Maude. Its top module is named `NuITP`. As any other Maude program it needs to be started after it is loaded into the system. The `NuITP.maude` file ends up with such a command so it is directly initialized. If the tool is stopped, for example by 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

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

Currently, there is no way to load Maude modules after the tool has been started — we can load scripts, as we will see in Section 2.1.3, but not files containing Maude modules. Therefore, you must either load your modules before the `NuITP.maude` file, or stop the tool, load the files using the `load` command, and then restart the tool (see above for how this can be done).

For example, you can initiate Maude specifying the modules as arguments of the `maude` command:

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

or start Maude and then load the files.

```
$ maude -allow-files

          \||||||||||||||||||/
        --- Welcome to Maude ---
          /||||||||||||||||||\
 Maude 3.3.1 built: Apr 13 2023 16:10:31
  Copyright 1997-2023 SRI International
        Thu Jul 6 09:33:00 2023
```

[5] 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`/`save`, and `export` commands.

```
Maude> load examples/peano+R.maude
Maude> load NuITP.maude
==========================================
erewrite in NuITP : init .

    ==================================
              NuITP (alpha 21)
          Inductive Theorem Prover
       for Maude Equational Theories
    ==================================
            Copyright 2021-2023
    Universitat Politècnica de València

NuITP>
```

Once the modules have been loaded into Maude, and 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 them is the chosen one. To select a module we can use the `set module` command.

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

For example,

```
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 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.[6] 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 trees (note that these commands do not have a final dot):

`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.

[6] 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.

1.5 A simple proof: proving 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 [7] for detailed descriptions of the inference rules, and Section 2 for a detailed information, concrete syntax, and examples on each command of the tool. Some additional examples can be found in Section 4.

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 metadata "1" ] .
  op s_ : Nat -> NzNat [ ctor metadata "2" ] .

  op _+_ : Nat Nat -> Nat [ metadata "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 `metadata` 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 [5]. See Section 1.3 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 the 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 3.3.1 built: Apr 13 2023 16:10:31
  Copyright 1997-2023 SRI International
        Thu Jul 6 09:33:00 2023
==========================================
erewrite in NuITP : init .

    ==================================
              NuITP (alpha 21)
           Inductive Theorem Prover
```

```
       for Maude Equational Theories
    ================================
            Copyright 2021-2023
    Universitat Politècnica de València

NuITP>
```

Then, we set the module as the current active one.

```
NuITP> set module PEANO+R .

  Module PEANO+R is now active.
```

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
  Skolem Ops:
    None
  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.[7]

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

  Generator Set Induction (GSI) applied to goal 0.

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

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

[7] Generator sets do not need to be explicitly given in every command requiring one. See Section 2.2 for information on how to define and use generator sets for the different sorts in your specifications.

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.

  Unproved goals:

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

  Total unproved 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.

1.6 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 1.5 is much simpler.

```
NuITP> set module PEANO+R .

  Module PEANO+R is now active.

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

  Initial goal set.

  Goal Id: 0
  Skolem Ops:
    None
  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 with Equality Predicate Simplification (GSI!)
  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.

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

  Initial goal set.

  Goal Id: 0
  Skolem Ops:
    None
  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 with Equality Predicate Simplification (GSI!)
  applied to goal 0.

  Goal Id: 0.1.1
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $2:Nat =(0 + $2:Nat)

  Goal Id: 0.2.1
  Skolem Ops:
    $3.Nat
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    ($3 + $2:Nat) = ($2:Nat + $3)
  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 with Equality Predicate Simplification (GSI!)
  applied to goal 0.1.1.

  Goals 0.1.1.1.1 and 0.1.1.2.1 have been proved.

  Unproved goals:

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

  Total unproved 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 with Equality Predicate Simplification (GSI!)
  applied to goal 0.2.1.

  Goals 0.2.1.1.1 and 0.2.1.2.1 have been proved.

  qed
```

An interesting thing about this proof is that the goals are discharged thanks to the application of non-executable hypotheses handled by a form of ordered rewriting that orients them using the given RPO (see [7]). Again, we can ask the tool to print all internal goals with the `show goals .` or `show goal <`*goal-id*`> .` commands.

1.8 Proving program equivalence

Using NuITP, we can prove that the `PEARNO+R` module presented in Section 1.5 and the module `PEANO+L` below are *equivalent*, that is, that they compute the same addition function. We can do so by proving in `PEARNO+R` the axioms in `PEANO+L` and vice versa.[8]

```
set include BOOL off .

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

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

  op _+_ : Nat Nat -> Nat [ metadata "3" ] .
```

[8]For a general notion of equivalence between equational programs and a justification of the proof method see: J. Meseguer, Lecture 14, Lectures Notes for CS 476, Fall 2022, University of Illinois at Urbana-Champaign, available at https://courses.grainger.illinois.edu/CS476/fa2022/#lecture-14-11th-oct.

```
  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
  Skolem Ops:
    None
  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 with Equality Predicate Simplification (GSI!)
  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.

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

  Initial goal set.

  Goal Id: 0
  Skolem Ops:
    None
  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 with Equality Predicate Simplification (GSI!)
  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.

1.9 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 2.4.

1.9.1 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 .` or `internalize as assoc .` commands) 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 1.8, 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) .
```

In this enriched module, we can now carry on proofs of associativity and commutativity of addition in one single step.[9]

[9]Note that the input has been broken into several lines to facilitate its readability. Any command input to

```
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
  Skolem Ops:
    None
  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 with Equality Predicate Simplification (GSI!) applied to
  goal 0.

  Goals 0.1.1 and 0.2.1 have been proved.

  qed
```

1.9.2 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 metadata "0"] .
    op s_ : Nat -> NzNat [ctor metadata "1"] .

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

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

    op _*_ : Nat Nat -> Nat [prec 40 metadata "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 must be entered in a single line (without carriage return).

```
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 of addition and internalize it.

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

  Initial goal set.

  Goal Id: 0
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    ($1:Nat + $2:Nat) =($2:Nat + $1:Nat)

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

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

  Goal Id: 0.1.1
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $2:Nat =(0 + $2:Nat)

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

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

  Generator Set Induction with Equality Predicate Simplification (GSI!) 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
  Skolem Ops:
    $3.Nat
  Executable Hypotheses:
```

```
    None
  Non-Executable Hypotheses:
    ($3 + $2:Nat) =($2:Nat + $3)
  Goal:
    s($2:Nat + $3) =(s $3 + $2:Nat)

  Total unproven goals: 1

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

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

  Goals 0.2.1.1.1 and 0.2.1.2.1 have been proved.

  qed

NuITP> internalize as comm .

  Proven goal has been internalized as comm axiom.
```

Then we do the same for associativity of addition.

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

  Initial goal set.

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

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

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

  Goals 0.1.1 and 0.2.1 have been proved.

  qed

NuITP> internalize as assoc .

  Proven goal has been internalized as assoc axiom.
```

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
  Skolem Ops:
    None
  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 with Equality Predicate Simplification (GSI!) applied to
  goal 0.

  Goals 0.1.1 and 0.2.1 have been proved.

  qed
```

1.9.3 Proving a program optimization

On the enriched module of Section 1.9.1, 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.[10]

```
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) .

  Initial goal set.

  Goal Id: 0
  Skolem Ops:
    None
  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.
```

[10] 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. Note also 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.

```
Goal 0.1 has been proved.

qed
```

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

2 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 2.1 we present the NuITP general commands. In Section 2.2, we explain the generator-set related commands. Section 2.4 explains the different internalization commands. In Section 2.5 we introduce the simplification commands, which require less user interaction. Finally, in Section 2.6 we introduce the induction commands, which may require more user interaction and add induction hypotheses to the current goal.

2.1 General NuITP commands

2.1.1 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 1.1.

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
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    s($1:Nat + $2:Nat) = (s $1:Nat + $2:Nat)
```

```
NuITP>
```

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

2.1.2 Information commands

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

- The `show module` command shows the currently active module.
- The `show log` command shows a raw log of the session.
- 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 goal` <*goal-id*> shows the goal with the specified identifier.
- The `show gensets` command shows all the generator sets defined by the user in the current session (see Section 2.2).
- 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.

Syntax

```
show module .
show log .
show goals .
show frontier .
show goal <goal-id> .
show gensets .
show gensets for <sort> .
show genset <gen-set-id> .
set verbose (on | off) .
```

Examples

```
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 .
```

2.1.3 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.

- 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`, and `tex` commands, which respectively either (i) save a minimal[11] 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 report of the current proof and save it in the specified file.[12]

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 report into the provided file. It is thus the user responsibility to provide a safe filename.

Syntax

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

Examples

```
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
```

[11] Where the effects of possible `undo` commands have been applied.
[12] Note that NuITP will automatically add a `.proof`, `.snap`, or `.tex` extension, respectively, to the provided file name if it is omitted.

Requirements

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

2.1.4 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

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

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

Example

```
NuITP> undo 0.1 .
```

```
NuITP> clear .
```

```
NuITP> clear all .
```

Requirements

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.

2.1.5 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

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

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

Example

```
NuITP> assert .
```

```
NuITP> assert 0.1 .
```

2.1.6 Help

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

```
NuITP> help
```

2.1.7 Quit

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

```
NuITP> quit
```

or

```
NuITP> q
```

2.2 Generator-sets commands

2.2.1 Adding a generator set

Syntax

`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.

Example

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

Requirements

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 type 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 set 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 set the one to be used by default, we may use the command in the following section.

2.2.2 Setting a generator set as default

Syntax

`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.

Example

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

Requirements

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

2.3 Simplification strategies commands

2.3.1 Adding a simplification strategy

Syntax

`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

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

Requirements

The specified strategy must consist of a list of NuITP simplification strategies separated by semicolons. By default, 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.

2.3.2 Setting a simplification strategy as default

Syntax

`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

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

Requirements

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

2.3.3 Applying a simplification strategy

Syntax

`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. In the case that no strategy identifier is given, the current default simplification strategy will be applied.

Example

```
NuITP> simplify 0 .
```

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

Requirements

If explicitly given, the simplification strategy with the corresponding identifier must have been previously defined.

2.4 Internalization commands

2.4.1 Internalization of a proven formula as equation or hypothesis

Syntax

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

Example

```
NuITP> internalize .
```

```
NuITP> internalize 0.1 .
```

See examples of use in Sections 1.9.1 and 1.9.3.

Requirements

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 each of the clauses in the goal will be internalized as different formulas:

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

$$\ldots$$

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

If one of the above clauses is a (possibly conditional) equation and can be oriented as a rewrite rule using the provided RPO, then it will be internalized as an (executable) equation. Otherwise, it will be added as a hypothesis, available for subsequent proofs.

2.4.2 Internalization of all intermediate proven goals

Syntax

`internalize all .`

Example

```
NuITP> internalize all .
```

Requirements

The proof of the current active goal has been successfully completed. Each of the intermediate subgoals proven along the proof of the latest goal that do not have any Skolen constant are added as equations or hypotheses. While in the `internalize .` command only the main goal is added, in the `internalize all .` command each intermediate goal is internalized, but the procedure is the same in both cases, each goal is check using the available RPO and one thing or the other is done depending on whether it can be oriented.

2.4.3 Internalization of a proven formula as an axiom

Syntax

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

Example

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

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

Requirements

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 $f(f(X, Y), Z) = f(X, f(Y, Z))$. In both cases, f must be a binary operator, all variables used in the goal (X, Y, Z) must be declared of the same sort S, which is the maximal sort of the sort component it belongs to, and among the declarations for the operator f there must be a declaration $f : S\ S \to S$ in the module. The operator f can be either in prefix or infix notation.

2.4.4 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

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

Example

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

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

Requirements

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

2.5 Simplification commands

In NuITP, simplification inference rules transform goals into simpler goals, sometimes proving them altogether. Since applying them is almost always advantageous, they are ideally suited to be automated by means of strategies. Furthermore, they require little to none 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.

2.5.1 Equality Predicate Simplification (EPS)

Syntax

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

where <*goal-id*> is the identifier of the goal that we want to simplify with the **EPS** rule.

Example

```
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.

See examples of use in Sections 1.5, 1.9.3, 4.1, 4.2, and 4.3.

2.5.2 Constructor Variant Unification Left (CVUL)

Syntax

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

where <*goal-id*> is the identifier of the goal that we want to simplify with the **CVUL** rule.

Example

```
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

The condition of the multiclause contains at least one equality $u = v$ such that is a Σ_1-formula, i.e., both u and v are defined in Σ_1 (see Section 1.1).

Recall that, here and in what follows, adding `!` after a command means applying `EPS` to each of the goals generated by the given command.

2.5.3 Constructor Variant Unification Failure Right (CVUFR)

Syntax

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

where <*goal-id*> is the identifier of the goal that we want to simplify with the **CVUFR** rule.

Example

```
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

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 $unif^{\Omega}_{\mathcal{E}_1}(u = v)$ is empty (see Section 1.1).

2.5.4 Substitution Left (SUBL)

Syntax

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

where <*goal-id*> is the identifier of the goal that we want to simplify with the **SUBL** rule.

Example

```
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

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 1.1).

2.5.5 Substitution Right (SUBR)

Syntax

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

where <*goal-id*> is the identifier of the goal that we want to simplify with the **SUBR** rule.

Example

```
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

The right-hand side Δ of the multiclause $\Gamma \rightarrow \Delta$ 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 than or equal to the sort of $\bar{x}$; (iv) u is not a Σ_1-term; and (v) the rest of the multiclause's right-hand side is not empty.

2.5.6 Narrowing Simplification (NS)

Syntax

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

where <*goal-id*> is the identifier of the goal that we want to simplify with the **NS** rule.

Example

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

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

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

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

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

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

```
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

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 1.1).

2.5.7 Clause Subsumption (CS)

Syntax

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

where <*goal-id*> is the identifier of the goal that we want to simplify with the **CS** rule.

Example

```
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

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

2.5.8 Inductive Congruence Closure (ICC)

The main purpose of the **ICC** rule is to try to discharge the goal $\Gamma \to \Lambda$ by rewriting simplification using H and $\mathcal{E}^{=}_{\overline{X}_U}$, either showing Γ unsatisfiable or proving $\overline{\Lambda}$ by assuming $\overline{\Gamma}$.

Syntax

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

where <*goal-id*> is the identifier of the goal that we want to simplify with the **ICC** rule.

Example

```
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

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

2.5.9 Variant Satisfiability (VARSAT)

Syntax

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

where <*goal-id*> is the identifier of the goal that we want to simplify with the **VARSAT** rule.

Example

```
NuITP> apply varsat to 0.1 .
```

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

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

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

Requirements

The clause of the goal is an Σ_1-formula and its negation is unsatisfiable in $T_{\mathcal{E}_1}$.

2.5.10 Equality (EQ)

Syntax

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

where <*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 `EQ!` version of the rule corresponds to applying **EQ** followed by **EPS**.

Examples

```
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 4.4.

Requirements

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.

2.6 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 (`*`)[13] in order to automatically simplify the rule's resulting subgoals.

2.6.1 Generator Set Induction (GSI)

Syntax

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

where < *goal-id* > is the identifier of the goal on which we want to apply the **GSI** rule, < *var* > is the variable on which we want to apply induction, < *gen-set-id* > is the identifier of a previously defined 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 variable's sort.

Examples

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

```
NuITP> apply gsi! to 0.1 on $1 .
```

```
NuITP> apply gsi* to 0.1 on $1 .
```

```
NuITP> apply gsi{MY-STRAT} to 0.1 on $1 .
```

```
NuITP> apply gsi to 0.1 on $1 with GEN-NAT .
```

```
NuITP> apply gsi! to 0.1 on $1 with GEN-NAT .
```

```
NuITP> apply gsi* to 0.1 on $1 with GEN-NAT .
```

```
NuITP> apply gsi{MY-STRAT} to 0.1 on $1 with GEN-NAT .
```

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

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

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

```
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 in Sections 1.5, 1.6, 1.7, 1.8, and 4.2.

Requirements

The goal's multiclause contains the variable on which we want to apply **GSI**. Furthermore, the generator set for the sort of the chosen variable should be a correct generator set. Checking the correctness of the generator set (a check which can be semi-automated using Maude's Sufficient Completeness Checker (SCC)) is the user's responsibility.[14]

2.6.2 Narrowing Induction (NI)

Syntax

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

where < *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

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

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

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

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

See examples of use in Sections 4.3.

[14] 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 [6]. 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.

Requirements

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.

2.6.3 Lemma Enrichment (LE)

Syntax

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

where < *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

```
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 4.4.

2.6.4 Split (SP)

Syntax

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

where < *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

```
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 + s(0) > 0 = true) \/ (N:Nat + s(0) <= 0 = true) .
```

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

```
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

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

2.6.5 Case (CAS)

Syntax

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

where $<$*goal-id*$>$ is the identifier of the goal to which we want to apply the **CAS** rule, $<$*var*$>$ is the variable or *Skolem* constant on which we want to apply cases, $<$*gen-set-id*$>$ is the identifier of a previously defined generator set, and $<$ *gen-set* $>$ is a set of generator terms separated by double semicolons that 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.

Example

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

```
NuITP> apply cas! to 0.1 on $1 .
```

```
NuITP> apply cas* to 0.1 on $1 .
```

```
NuITP> apply cas{MY-STRAT} to 0.1 on $1 .
```

```
NuITP> apply cas to 0.1 on $1 with GEN-NAT .
```

```
NuITP> apply cas! to 0.1 on $1 with GEN-NAT .
```

```
NuITP> apply cas* to 0.1 on $1 with GEN-NAT .
```

```
NuITP> apply cas{MY-STRAT} to 0.1 on $1 with GEN-NAT .
```

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

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

```
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) .
```

Requirements

The goal's multiclause contains the variable on which we want to apply the **CAS** rule. As already mentioned for the `gsi` command, in this alpha 28 version of the NuITP, checking the correctness of the generator set is the user's responsibility.

2.6.6 Variable Abstraction (VA)

Syntax

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

where *<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

```
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

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.

2.6.7 Cut (CUT)

Syntax

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

where *<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

```
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) .
```

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

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.

3 Simplification 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, a `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 1.6 or 1.7.

3.1 Defining more complex strategies

As of NuITP alpha 28, 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-subr`	`try-varsat`	`try-cas`
`try-cvul`	`try-ns`	`try-gnd`	
`try-cvufr`	`try-cs`	`try-rst`	
`try-subl`	`try-icc`	`try-gsi`	

Note that all the above basic strategies include a final `try-eps` internally, except for the `try-eps` itself. Also note that NuITP automatically provides a default simplification strategy with identifier `NuITP`, as shown in the following example:

```
    ==================================
                 NuITP
          Inductive Theorem Prover
       for Maude Equational Theories
       (alpha 28 built Apr 5th 2024)
    ==================================
           Copyright 2021-2023
    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 the account how the new frontier of the current proof evolves via parallel 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., the goal has only one variable (or Skolem constant) and a default generator set is defined for its sort.

Eventually, NuITP will provide facilities for the definition of new strategies using Maude's strategy language [3] and some additional facilities. 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 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.

4.1 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.[15]

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

  op 0 : -> Zero [ ctor metadata "1" ] .
  op s : Nat -> NzNat [ ctor metadata "2" ] .
```

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

```
  sorts Pair UPair .

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

  vars N M : Nat .

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

  op _*_ : Nat Nat -> Nat [ assoc comm metadata "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 metadata "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[16] 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
  Skolem Ops:
    None
  Executable Hypotheses:
    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
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
```

[16] All NuITP commands must be written in a single line (see Section 5.1 for more info).

```
    (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.

4.2 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 metadata "1" ] .
  op s : Nat -> Nat [ ctor metadata "2" ] .

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

  op _@_ : List List -> List [ metadata "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 .
```

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
  Skolem Ops:
    None
  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
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    (nil @ ($2:List @ $3:List)) = ((nil @ $2:List) @ $3:List)

  Goal Id: 0.2
  Skolem Ops:
    $4.List
    $5.Nat
  Executable Hypotheses:
    (($4 @ $2:List) @ $3:List) => ($4 @ ($2:List @ $3:List))
  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.

  Unproved goals:

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

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

  Generator Set Induction with Equality Predicate Simplification (GSI!) 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.

4.3 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 metadata "1" ] .

  op rev : List -> List [ metadata "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 .
```

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
  Skolem Ops:
    None
  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
  Skolem Ops:
    $3.Elt
    $4.Elt
    $5.List
  Executable Hypotheses:
    rev($5 $4) => $4 rev($5)
  Non-Executable Hypotheses:
    None
  Goal:
    ($4 rev($3 $5)) = rev($5 $4) $3

  Goal Id: 0.2
  Skolem Ops:
    None
  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.

  Unproved goals:

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

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

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

  Goals 0.1 and 0.2 have been proved.

  qed
```

In the following, we will use the extended, ! versions of the NuITP 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.

4.4 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 metadata "1" ] .
  op s : Nat -> NzNat [ ctor metadata "2" ] .

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

  op _*_ : Nat Nat -> Nat [ assoc comm metadata "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:

```
NuITP> set module PEANO+RAxR .
```

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
  Skolem Ops:
    None
  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 with 0 ;; s(0) ;; s(s(Z:Nat)) .

  Case (CAS) applied to goal 0.

  Goal Id: 0.1
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    s(0) = 0 * 0 -> 0 = s(0)

  Goal Id: 0.2
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    s(0) = s(0) * s(0) -> s(0) = s(0)

  Goal Id: 0.3
  Skolem Ops:
    None
  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.

  Unproved goals:

  Goal Id: 0.2
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    s(0) = s(0) * s(0) -> s(0) = s(0)

  Goal Id: 0.3
  Skolem Ops:
```

```
    None
  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 unproved goals: 2

NuITP> apply eps to 0.2 .

  Equality Predicate Simplification (EPS) applied to goal 0.2.

  Goal 0.2.1 has been proved.

  Unproved goals:

  Goal Id: 0.3
  Skolem Ops:
    None
  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 unproved 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
  Skolem Ops:
    None
  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
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    ($4:Nat + $3:Nat) = $3:Nat + $4:Nat
```

```
  Goal Id: 0.3.1.2
  Skolem Ops:
    None
  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 .

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

  Goal Id: 0.3.1.2.1.1
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    ($4:Nat + $3:Nat) = $3:Nat + $4:Nat
  Goal:
    0 = s(s($2:Nat + s($2:Nat) * s($2:Nat))) + $2:Nat -> false
```

Note that any ground instance of the goal's condition will be false, since the equality that states that `0 = s(s($2:Nat + s($2:Nat) * s($2:Nat))) + $2:Nat` will never be true in our example. Also note that the subterm `s($2:Nat) * s($2:Nat)` cannot be reduced using the equations in our theory. We can however apply **GSI!** with the usual generator set for natural numbers in Peano notation and "get" the extra `s` symbol we are missing to be able to reduce such terms with the original equations as follows:

```
NuITP> apply gsi! to 0.3.1.2.1.1 on $2 with 0 ;; s(0) ;; s(s(W:Nat)) .

  Generator Set Induction with Equality Predicate Simplification (GSI!)
  applied to goal 0.3.1.2.1.1.

  Goals 0.3.1.2.1.1.1.1, 0.3.1.2.1.1.2.1 and 0.3.1.2.1.1.3.1 have been proved.

  Unproved goals:

  Goal Id: 0.3.1.1
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    ($4:Nat + $3:Nat) = $3:Nat + $4:Nat

  Total unproved goals: 1
```

At this point, the only remaining goal is the lemma we introduced with the **LE** rule, which

needs to be proved[17] 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.

4.5 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
  pr TRUTH-VALUE .

  sorts Zero NzNat Nat .
  subsorts Zero NzNat < Nat .

  op 0 : -> Zero [ ctor metadata "1" ] .
  op 1 : -> NzNat [ ctor metadata "2" ] .

  op _>_ : Nat Nat -> Bool [ metadata "3" ] .
  eq (X:Nat + X':NzNat) > X:Nat = true [ variant ] .
  eq X:Nat > (X:Nat + Y:Nat) = false [ variant ] .

  op _+_ : Nat Nat -> Nat [ assoc comm metadata "4" ] .
  op _+_ : Nat NzNat -> NzNat [ assoc comm metadata "4" ] .
  op _+_ : NzNat Nat -> NzNat [ assoc comm metadata "4" ] .
  op _+_ : NzNat NzNat -> NzNat [ ctor assoc comm metadata "4" ] .
  eq X:Nat + 0 = X:Nat [ variant ] .

  op _*_ : Nat Nat -> Nat [ assoc comm metadata "5" ] .
  op _*_ : NzNat NzNat -> NzNat [ assoc comm metadata "5" ] .
  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
```

```
NuITP> set module PRESBURGER&MULT .
```

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
  Skolem Ops:
    None
  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:

[17] Two alternative proofs of the commutativity of addition can be found in Section 1.7.

```
NuITP> apply gsi! to 0 on $1 with 0 ;; 1 ;; 1 + X:NzNat .

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

  Goal Id: 0.1.1
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    0 = $2:Nat * $3:NzNat -> 0 = $2:Nat

  Goal Id: 0.2.1
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $3:NzNat = $2:Nat * $3:NzNat -> 1 = $2:Nat

  Goal Id: 0.3.1
  Skolem Ops:
    $4.NzNat
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    $3:NzNat = $3:NzNat * $2:Nat -> 1 = $2:Nat
    ($4 * $3:NzNat) = $3:NzNat * $2:Nat -> $4 = $2:Nat
  Goal:
    ($2:Nat * $3:NzNat) = $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 with Equality Predicate Simplification (CAS!) applied to goal 0.1.1.

  Goal 0.1.1.2.1 has been proved.

  Goal Id: 0.1.1.1.1
  Skolem Ops:
    None
  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 0.1.1.1.1 on $3:NzNat * $4:NzNat .

  Variable Abstraction with Equality Predicate Simplification (VA!)
  applied to goal 0.1.1.1.1.

  Goal Id: 0.1.1.1.1.1.1
```

```
  Skolem Ops:
    None
  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 `0.1.1.1.1.1.1` ready to be simplified by means of the **CVUL** simplification rule[18], since the equality `0 = $5:NzNat` will never be true, which will falsify the condition proving the premise:

```
NuITP> apply cvul to 0.1.1.1.1.1.1 .

  Constructor Variant Unification Left (CVUL) applied to goal 0.1.1.1.1.1.1.

  Goal 0.1.1.1.1.1.1.1 has been proved.

  Unproved goals:

  Goal Id: 0.2.1
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $3:NzNat = $2:Nat * $3:NzNat -> 1 = $2:Nat

  Goal Id: 0.3.1
  Skolem Ops:
    $4.NzNat
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    $3:NzNat = $3:NzNat * $2:Nat -> 1 = $2:Nat
    ($4 * $3:NzNat) = $3:NzNat * $2:Nat -> $4 = $2:Nat
  Goal:
    ($2:Nat * $3:NzNat) = $3:NzNat + $4 * $3:NzNat -> $2:Nat = $4 + 1

  Total unproved goals: 2
```

Now we try to prove Goal `0.2` 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 with Equality Predicate Simplification (GSI!)
  applied to goal 0.2.1.

  Goal Id: 0.2.1.1.1
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $3:NzNat = $3:NzNat * $4:NzNat -> 1 = $4:NzNat

  Goal Id: 0.2.1.2.1
```

[18] Note that the new abstraction variable uses the goal identifier to avoid undesirable clashes.

```
  Skolem Ops:
    None
  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.

  Unproved goals:

  Goal Id: 0.3.1
  Skolem Ops:
    $4.NzNat
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    $3:NzNat = $3:NzNat * $2:Nat -> 1 = $2:Nat
    ($4 * $3:NzNat) = $3:NzNat * $2:Nat -> $4 = $2:Nat
  Goal:
    ($2:Nat * $3:NzNat) = $3:NzNat + $4 * $3:NzNat -> $2:Nat = $4 + 1

  Goal Id: 0.2.1.1.1
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    $3:NzNat = $3:NzNat * $4:NzNat -> 1 = $4:NzNat

  Total unproved goals: 2
```

and **CAS!** to Goal `0.2.1.1.1` but, this time, on variable `$4:NzNat`:

```
NuITP> apply cas! to 0.2.1.1.1 on $4 with 1 ;; 1 + Y:NzNat .

  Case with Equality Predicate Simplification (CAS!) applied to goal 0.2.1.1.1.

  Goals 0.2.1.1.1.1.1 and 0.2.1.1.1.2.1 have been proved.

  Unproved goals:

  Goal Id: 0.3.1
  Skolem Ops:
    $4.NzNat
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    $3:NzNat = $3:NzNat * $2:Nat -> 1 = $2:Nat
    ($4 * $3:NzNat) = $3:NzNat * $2:Nat -> $4 = $2:Nat
  Goal:
    ($2:Nat * $3:NzNat) = $3:NzNat + $4 * $3:NzNat -> $2:Nat = $4 + 1
```

```
  Total unproved 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 with Equality Predicate Simplification (CAS!) 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
  Skolem Ops:
    $4.NzNat
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    $3:NzNat = $3:NzNat * $2:Nat -> 1 = $2:Nat
    ($4 * $3:NzNat) = $3:NzNat * $2:Nat -> $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.

4.6 Reversing Palindromes

Consider the following conditional equational theory, which extends the theory of Section 4.3 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:

```
fmod REVERSING-PALINDROMES is
  pr TRUTH-VALUE .
  sorts Elt List .
  subsort Elt < List .

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

  op _=e=_ : List List -> Bool [ metadata "2" ] .
  eq L:List =e= L:List = true .
  eq L:List =e= Q:List = false [ owise ] .

  op rev : List -> List [ metadata "3" ] .
  eq rev(X:Elt) = X:Elt .
  eq rev(X:Elt L:List) = rev(L:List) X:Elt .

  op pal : List -> Bool [ metadata "4" ] .
  eq pal(X:Elt) = true .
  eq pal(X:Elt X:Elt) = true .
```

```
  eq pal(X:Elt Q:List X:Elt) = pal(Q:List) .
  ceq pal(X:Elt Y:Elt) = false if (X:Elt =e= Y:Elt) = false .
  ceq pal(X:Elt Q:List Y:Elt) = false if (X:Elt =e= Y:Elt) = false .
endfm
```

```
NuITP> set module REVERSING-PALINDROMES .
```

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
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    true = pal($1:List) -> $1:List = rev($1:List)
```

First, we introduce an auxiliary lemma `rev(Q:List Y:Elt) = Y:Elt rev(Q:List)` in our current proof, which we already proved to be true in the reversing lists example of Section 4.3:

```
NuITP> apply le! to 0 with rev(Q:List Y:Elt) = Y:Elt rev(Q:List) .

  Lemma Enrichment with Equality Predicate Simplification (LE!) applied to goal 0.

  Goal Id: 0.1
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    rev($2:List $3:Elt) = $3:Elt rev($2:List)

  Goal Id: 0.2
  Skolem Ops:
    None
  Executable Hypotheses:
    rev($2:List $3:Elt) => $3:Elt rev($2:List)
  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 with Equality Predicate Simplification (NI!)
  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
  Skolem Ops:
    $4.Elt
    $5.List
  Executable Hypotheses:
    rev($2:List $3:Elt) => $3:Elt rev($2:List)
    true = pal($5) -> rev($5) => $5
  Non-Executable Hypotheses:
    None
  Goal:
    true = pal($5) -> $5 = rev($5)
```

and then applying clause subsumption (**CS**) to the remaining goal. 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:

```
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.

  Unproved goals:

  Goal Id: 0.1
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    rev($2:List $3:Elt) = $3:Elt rev($2:List)

  Total unproved goals: 1
```

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 Troubleshooting

In the following, we describe some requirements and common problems that can arise while running NuITP and how to avoid them.

5.1 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.
```

5.2 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`, `save` and `export` commands, but should not affect the remaining features of NuITP.

5.3 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 28 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*[19] filename that will not result in a potential risk.

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 3.2.1). Technical report, SRI International Computer Science Laboratory, 2022. Available at: http://maude.cs.illinois.edu.

[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*. Springer, 2007.

[4] H. Comon-Lundh and S. Delaune. The Finite Variant Property: How to Get Rid of Some Algebraic Properties. In *Proceedings of the 16th International Conference on Rewriting Techniques and Applications (RTA 2005)*, volume 3467 of *Lecture Notes in Computer Science*, pages 294–307. Springer, 2005.

[5] R. Gutiérrez, J. Meseguer, and S. Skeirik. The maude termination assistant. In *Preproceedings of International Workshop on Rewriting Logic and its Applications (WRLA)*, 2018.

[6] J. Hendrix, J. Meseguer, and H. Ohsaki. A sufficient completeness checker for linear order-sorted specifications modulo axioms. In *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 151–155. Springer, 2006.

[7] J. Meseguer and S. Skeirik. Inductive Reasoning with Equality Predicates, Contextual Rewriting and Variant-Based Simplification. Submitted for publication.

[19] Remember that NuITP automatically adds the proper extension to the filename if it is omitted.