NuITP

alpha 12a

Inductive Theorem Prover
for Maude Equational Theories

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

November 16, 2022

Contents

1 Getting started

1.1 Running NuITP

NuITP runs with the latest version of the Maude System (version 3.2.1)[1], which you can download from the Maude's website at http://maude.cs.illinois.edu.

The current version of NuITP is distributed as a zip file consisting of two files, namely `NuITP.maude` and `aacrpo.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:

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

          \||||||||||||||||||/
        --- Welcome to Maude ---
          /||||||||||||||||||\
 Maude 3.2.1 built: Feb 21 2022 18:24:38
  Copyright 1997-2022 SRI International
        Mon Oct  3 11:47:25 2022
==========================================
erewrite in NuITP : init .

              NuITP
     Inductive Theorem Prover
  for Maude Equational Theories
            alpha 12a

NuITP>
```

NuITP is an interactive tool, with a number of commands you will 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.[2]

Before we go with the commands, 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 initiated. If the tool is stopped, for example using 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 .
```

You will carry on your proofs on specifications that are assumed previously loaded in Maude. Currently, there is no way to load files after the tool has been started. 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 — to restart the tool after loading some module, you must set the `NuITP` module as current module and then initiate it with `erew init` as explained above.

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

[1]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 rest of 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.

[2]Most of the 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 -allow-files examples/peano+R.maude NuITP.maude
```

or start Maude and then load the files.

```
$ maude -allow-files

          \||||||||||||||||||/
        --- Welcome to Maude ---
          /||||||||||||||||||\
 Maude 3.2.1 built: Feb 21 2022 18:24:38
  Copyright 1997-2022 SRI International
        Mon Oct  3 11:47:25 2022
Maude> load examples/peano+R.maude
Maude> load NuITP.maude
==========================================
erewrite in NuITP : init .

              NuITP
     Inductive Theorem Prover
  for Maude Equational Theories
            alpha 12a

NuITP>
```

Once your modules have been loaded into Maude, and the NuITP tool is ready to execute commands, we can begin our interaction. We can carry out inductive proofs on any module loaded in Maude, but we need to tell the theorem prover which of 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 the 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.[3] The following commands are useful to manage these goals:

`set goal` *<goal>* `.`: sets the specified goal as 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.

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

`export` *<file-name>*: saves a report on the current proof in the specified file.

[3] 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 will prove the top goal.

1.2 Preliminaries and Assumptions

NuITP makes use of state-of-the-art advances in fields such as variant unification, narrowing, rewriting with strategies, and meta-interpreters, all of which have been gradually incorporated into the Maude system. Therefore, for a full compatibility, it is recommended to run NuITP in the last available version of the Maude interpreter (i.e., version 3.2.1). Otherwise, the requirements are those of the Maude's interpreter itself.

The Maude specifications currently handled by NuITP are functional modules of the form **fmod** $(\Sigma, E \cup B)$ **endfm**, with E a set of ground convergent equations modulo B, where B a set of axioms, which can be any combination of associativity (A), commutativity (C) or identity (U) axioms. Furthermore, the module is sufficiently complete with respect to a subsignature of constructors Ω, and, except for some axioms $B_\Omega \subseteq B$ among such constructors, there are no equations in E identifying constructor terms. That is, constructors are *free* modulo B_Ω.

1.3 The `ctor` and `variant` attributes

NuITP implements the theory described in [5], 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, the subtheory that has the Finite Variant Property (FVP) [3], and the original theory itself. 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. The NuITP relies on the user to clearly specify these two subtheories by means of Maude's `ctor` declarations and `variant` equation attributes (for more info, see Section 4.4.3 and [2, Chapter 14]). Specifically, all constructor symbols need to be declared with the `ctor` attribute and equations in E_1 should be unconditional and must be declared with the `variant` attribute *iif* E_1 has the FVP modulo B_1. The remaining equations in $E \setminus E_1$, which do not have the FVP, should not use the `variant` attribute. Equations in $E \setminus E_1$ can be conditional; but they should *not* use any built-in features such as the `==` equality predicate or the `owise` attribute (see [2] for more info on Maude's variant equations requirements).

1.4 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[4] B). The NuITP can then use this RPO order $\prec$ to automatically identify and orient a rules 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 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 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:

[4]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 tranforms the equations E to match without identity axioms.

```
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 its recursive equations). Then, a suitable RPO order for this theory making it terminating is $\texttt{0} \prec \texttt{s} \prec \texttt{+}$. Thus, 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. 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.

1.5 Clauses and multiclauses

The formulas that can be shown to be inductive theorems of a given Maude functional mpdule by the 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,

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

1.6 A simple proof: associativity of addition

In this section, we illustrate the basic operation of the theorem prover. As already stated, in addition to the commands related to the management of the proof tree itself, the prover provides several commands, which basically apply corresponding inference rules. Please, see [5] for detailed descriptions of the inference rules, and Section 2 for a detailed info, concrete syntax and examples on each command of the tool. Some additional examples can be found in Section 3.

All these commands have the form

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

which basically apply a given inference rule on a specific goal. For example, we can request the application of the `EPS` rule for equational predicate simplification on a specific pending goal, say `0.1`, by giving the command:

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

Let us see now some of these commands in action. To do so, let us begin with something simple, the associativity of natural numbers defined using the Peano notation. Let us use the following specification of numbers.

```
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 [4]. See Section 1.4 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. However, if the user needs Boolean values, the module `TRUH-VALUE` imports just the truth values without creating 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 3.2.1 built: Feb 21 2022 18:24:38
  Copyright 1997-2022 SRI International
        Mon Oct  3 11:47:25 2022
==========================================
erewrite in NuITP : init .

              NuITP
     Inductive Theorem Prover
  for Maude Equational Theories
            alpha 12a

NuITP>
```

Then, we set the module as current 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:
    (X:Nat + (Y:Nat + Z:Nat)) = ((X:Nat + Y:Nat) + Z:Nat)
```

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 Z, using the generator set given by 0 and 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.

```
NuITP> apply gsi to 0 on Z:Nat 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:
    (X:Nat + (Y:Nat + 0)) = ((X:Nat + Y:Nat) + 0)

  Goal Id: 0.2
  Skolem Ops:
    K.Nat
  Executable Hypotheses:
    ((X:Nat + Y:Nat) + K) => (X:Nat + (Y:Nat + K))
  Non-Executable Hypotheses:
    None
  Goal:
    (X:Nat + (Y:Nat + s K)) = ((X:Nat + Y:Nat) + s K)
```

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 discarded by equality predicate simplification 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:
    K.Nat
  Executable Hypotheses:
    ((X:Nat + Y:Nat) + K) => (X:Nat + (Y:Nat + K))
  Non-Executable Hypotheses:
    None
  Goal:
    (X:Nat + (Y:Nat + s K)) = ((X:Nat + Y:Nat) + s K)
```

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 on such fact.

1.7 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 combine them 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 1.6 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:
    (X:Nat + (Y:Nat + Z:Nat)) = ((X:Nat + Y:Nat) + Z:Nat)

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

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

  Goals 0.1 and 0.2 have been proved.

  qed
```

1.8 Commutativity of addition

If you have completed the proof of the commutativity of addition before, you know that we will need a lemma to complete the proof. But we cannot assume such things when we are facing a proving task. We present two alternative proofs for it. In Section 1.8.1 we complete the proof without using any previous knowledge. Then, in Section 1.8.2 we will present a more direct one in which by timely introducing the right lemma we will get the goal proved in a smaller number of steps.

1.8.1 Proving Commutativity of Addition the Hard Way

We are using our PEANO+R module, so we can begin by setting it and 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:
    (X:Nat + Y:Nat) = (Y:Nat + X:Nat)
```

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

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

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

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

  Goal Id: 0.2
  Skolem Ops:
    K.Nat
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    (K + Y:Nat) = (Y:Nat + K)
  Goal:
    s (Y:Nat + K) = (s K + Y:Nat)
```

Nothing new so far, we got the expected goals. We can try to solve them by using induction again. Let us begin with Goal `0.1`.

```
NuITP> apply gsi! to 0.1 on Y:Nat with 0 ;; s K:Nat .

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

  Goals 0.1.1 and 0.1.2 have been proved.

  Unproved goals:

   Goal Id: 0.2
  Skolem Ops:
    K.Nat
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    (K + Y:Nat) = (Y:Nat + K)
  Goal:
    s(Y:Nat + K) = (s K + Y:Nat)
```

The goal has automatically been discharged, and we are left with Goal `0.2`.

```
  NuITP> apply gsi! to 0.2 on Y:Nat with 0 ;; s K:Nat .

    Generator Set Induction with Equality Predicate Simplification (GSI!)
    applied to goal 0.2.

    Goal 0.2.2 has been proved.

    Goal Id: 0.2.1
    Skolem Ops:
      K.Nat
    Executable Hypotheses:
      None
    Non-Executable Hypotheses:
      (K + Y:Nat) = (Y:Nat + K)
    Goal:
      K = (0 + K)
```

We are almost done, but not quite: we are left with a goal that cannot be discharged by equational simplification. In fact, it is a particular case of Goal `0.1` that we proved before. Even though it was proved, the NuITP did not add it as a proved lemma to our theory. We need to add it explicitly as a lemma using the `LE` rule:

```
  NuITP> apply le to 0.2.1 with (Y:Nat = (0 + Y:Nat)) .

    Lemma Enrichment (LE) applied to goal 0.2.1.

    Goal Id: 0.2.1.1
    Skolem Ops:
      None
    Executable Hypotheses:
      None
    Non-Executable Hypotheses:
      None
    Goal:
      Y:Nat = (0 + Y:Nat)

    Goal Id: 0.2.1.2
    Skolem Ops:
      K.Nat
    Executable Hypotheses:
      (0 + Y:Nat) => Y:Nat
    Non-Executable Hypotheses:
      (K + Y:Nat) = (Y:Nat + K)
    Goal:
      K = (0 + K)
```

Of course, now we have our pending goal, and an additional goal for the lemma. With the executable hypotheses we can discharge Goal `0.2.1.2` by equational simplification.

```
  NuITP> apply eps to 0.2.1.2 .

    Equality Predicate Simplification (EPS) applied to goal 0.2.1.2.
    Goal 0.2.1.2.1 has been proved.

    Unproved goals:

    Goal Id: 0.2.1.1
    Skolem Ops:
      None
    Executable Hypotheses:
      None
    Non-Executable Hypotheses:
      None
    Goal:
      Y:Nat = (0 + Y:Nat)
```

Even though the lemma was not yet proven, we were able to use it. Of course, we need to discharge it to complete the proof.

```
NuITP> apply gsi! to 0.2.1.1 on Y:Nat with 0 ;; s K:Nat .

  Generator Set Induction with Equality Predicate Simplification (GSI!)
  applied to goal 0.2.1.1.

  Goals 0.2.1.1.1 and 0.2.1.1.2 have been proved.

  qed
```

In the following section we present an alternative proof in which, by introducing the required lemma sooner, we can complete the proof in fewer steps.

1.8.2 Proving Commutativity of Addition: An Easier Way

As usual, we begin by setting out module and goal.

```
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:
    (X:Nat + Y:Nat) = (Y:Nat + X:Nat)
```

Since we anticipate that left identity of addition will help in our proof, we introduce it as a lemma.

```
NuITP> apply le to 0 with (Y:Nat = (0 + Y:Nat)) .

  Lemma Enrichment (LE) applied to goal 0.

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

  Goal Id: 0.2
  Skolem Ops:
    None
  Executable Hypotheses:
    (0 + Y:Nat) => Y:Nat
  Non-Executable Hypotheses:
    None
  Goal:
    (X:Nat + Y:Nat) = (Y:Nat + X:Nat)
```

The executable hypothesis will now be used when needed. We may proceed by applying induction on the main goal.

```
NuITP> apply gsi! to 0.2 on X:Nat with 0 ;; s K:Nat .

  Generator Set Induction with Equality Predicate Simplification (GSI!)
  applied to goal 0.2.

  Goal 0.2.1 has been proved.

  Goal Id: 0.2.2
  Skolem Ops:
    K.Nat
  Executable Hypotheses:
    (0 + Y:Nat) => Y:Nat
  Non-Executable Hypotheses:
    (K + Y:Nat) = (Y:Nat + K)
  Goal:
    s(Y:Nat + K) = (s K + Y:Nat)
```

And again on Goal `0.2.2`.

```
NuITP> apply gsi! to 0.2.2 on Y:Nat with 0 ;; s K:Nat .

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

  Goals 0.2.2.1 and 0.2.2.2 have been proved.

  Unproved goals:

  Goal Id: 0.1
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    Y:Nat = (0 + Y:Nat)
```

And with one more application of `gsi!` we prove the only remaining goal:

```
NuITP> apply gsi! to 0.1 on Y:Nat with 0 ;; s K:Nat .

  Generator Set Induction with Equality Predicate Simplification (GSI!)
  applied to goal 0.1.
  Goals 0.1.1 and 0.1.2 have been proved.

  qed
```

1.9 Program equivalence

Using the NuITP, we can prove that the PEARNO+R module presented in Section 1.6 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.[5]

```
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" ] .
  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:
    (Y:Nat = (0 + Y:Nat)) /\ s(X:Nat + Y:Nat) = (s X:Nat + Y:Nat)

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

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

  Goals 0.1 and 0.2 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.
```

[5]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.

```
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:
    (Y:Nat = (Y:Nat + 0)) /\ s(Y:Nat + X:Nat) = (Y:Nat + s X:Nat)

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

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

  Goals 0.1 and 0.2 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.

1.10 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.

1.10.1 Proving commutativity and associativity of addition in Two Steps

In this enriched module PEANO+LR, the proofs of the commutativity and associativity properties of addition are straightforward.

```
NuITP> set module PEANO+LR .

  Module PEANO+LR is now active.

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:
    ((X:Nat + Y:Nat) = (Y:Nat + X:Nat)) /\ (X:Nat + (Y:Nat + Z:Nat)) =
    ((X:Nat + Y:Nat) + Z:Nat)

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

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

  Goal 0.1 has been proved.

  Goal Id: 0.2
  Skolem Ops:
    K.Nat
  Executable Hypotheses:
    ((X:Nat + K) + Z:Nat) => (X:Nat + (K + Z:Nat))
  Non-Executable Hypotheses:
    (K + X:Nat) = (X:Nat + K)
  Goal:
    (K + X:Nat) = (X:Nat + K)
```

As a result of the `gsi!` rule, the hypothesis coming from the application of the induction rule on the associativity goal has been oriented, and has been directly used for simplification. However, the hypothesis coming from the commutativity goal could not be oriented, and is left as a non-executable hypothesis. Goal `0.2` matches it perfectly, and that is why by the application of the **CS** rule in the last step, the goal is discharged, and the proof is completed.

```
  NuITP> apply cs to 0.2 .

    Clause Subsumption (CS) applied to goal 0.2.

    Goal 0.2.1 has been proved.

    qed
```

1.10.2 Program optimization

The enriched module `PEANO-LR` allows proving different kinds of goals without much effort. In the following interaction, 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.

```
  NuITP> set module PEANO+LR .

    Module PEANO+LR is now active.

  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 (X:Nat + Y:Nat) = (s s X:Nat + s s Y:Nat)) /\ ((s X:Nat + s Y:Nat) =
      (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)

  NuITP> apply eps to 0 .

    Equality Predicate Simplification (EPS) applied to goal 0.

    Goal 0.1 has been proved.

    qed
```

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. In Section 2.1 we present general commands. In Section 2.2 we introduce the simplification commands, which require less user interaction. In Section 2.3 we introduce the induction commands, which may require more user interaction and add hypothesis to the current goal.

2.1 General NuITP commands

2.1.1 The `set` command

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 1.2.

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: 0
  Skolem Ops:
    None
  Hypotheses:
    None
  Clause:
    s(X:Nat + Y:Nat) = (s(X:Nat) + Y:Nat)
```

```
NuITP>
```

Some 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 The `show` 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-identifier*> shows the goal with the specified identifier.

Syntax

```
show module .
show log .
show goals .
show frontier .
show goal  <goal-identifier>  .
```

Examples

```
NuITP> show module .
```

```
NuITP> show log .
```

```
NuITP> show goals .
```

```
NuITP> show frontier .
```

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

2.1.3 Load, save and export of proofs

The `load`, `save`, and `export` commands allow loading and saving proof scripts and proof trees, as well as generating proof reports. Note that these commands do not have a final dot.

- The `load` command loads (and executes) a proof script, which helps to automatize proofs.
- The counterpart of `load` is the `save` command, which saves a minimal[6] proof script of the current proof in the specified file.
- Finally, the `export` creates a LaTeX report of the current proof and saves it in the specified file[7].

Beware that, as in the current alpha 12a, NuITP does **not** ask for confirmation on the provided name when saving either the session script or the LaTeX report into the provided file. It is the user responsibility to provide a safe file name.

[6]Where the effects of possible `undo` commands have been applied.
[7]Note that NuITP will automatically add a `.tex` extension to the provided file name.

Syntax

```
load <file-name>
save <file-name>
export <file-name>
```

Examples

```
NuITP> load scripts/example
```

```
NuITP> save scripts/sessionScript
```

```
NuITP> export reports/proofReport
```

Requirements

Maude has to be started with the `-allow-files` flag.

2.1.4 Undo

The `undo` command, as its name indicates, undoes the effect of any 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.

Syntax

```
undo <goal-identifier> .
```

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

Example

```
NuITP> undo 0.1 .
```

Requirements

By definition, *undoable* goals 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.

2.1.5 Help

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

```
NuITP> help
```

2.1.6 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 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.

2.2.1 Equality Predicate Simplification (EPS)

Syntax

`apply eps to` *<goal-identifier>* `.`

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

Example

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

See examples of use in Sections 1.6, 1.8.1, 1.10.2, 3.1, 3.2, and 3.3.

2.2.2 Constructor Variant Unification Left (CVUL)

Syntax

`apply cvul to` *<goal-identifier>* `.`

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

Example

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

Requirements

The condition of the multiclause contains at least one equality $u = v$ such that both u and v are defined in Σ_1 (see Section 1.3).

2.2.3 Constructor Variant Unification Failure Right (CVUFR)

Syntax

`apply cvufr to` *<goal-identifier>* `.`

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

Example

```
NuITP> apply cvufr 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; and (iii) the set of constructor variant unifiers $unif^{\Omega}_{\mathcal{E}_1}(u = v)$ is empty (see Section 1.3).

2.2.4 Substitution Left (SUBL)

Syntax

`apply subl to` <*goal-identifier*> `.`

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

Example

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

Requirements

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

2.2.5 Substitution Right (SUBR)

Syntax

`apply subr to` <*goal-identifier*> `.`

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

Example

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

Requirements

The right-hand side of the multiclause 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.

2.2.6 Narrowing Simplification (NS)

Syntax

`apply ns to` <*goal-identifier*> `.`

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

Example

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

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 is also not a Σ_1 term; (iii) the terms $\vec{v}$ are constructor terms; and (iv) u is a Σ_1 term (see Section 1.3).

2.2.7 Clause Subsumption (CS)

Syntax

`apply cs to` *<goal-identifier>* `.`

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

Example

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

See an example of use in Section 1.10.1.

Requirements

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

2.2.8 Equality (EQ)

Syntax

`apply eq`[`!`] `to` *<goal-identifier>* `with` *<hypothesis>* [`sub` *<substitution>*] `.`

where *<goal-identifier>* 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!** versions of the rule correspond 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 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) .
```

See an example of use in Section 3.4.

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.

2.3 Induction commands

In the following, we show the induction commands available in the current version of NuITP. Note that all induction commands have two options, the simple version that applies the specified rule and the extended `!` version that applies the rule likewise, but followed by **EPS** in order to automatically simplify the rule's resulting subgoals.

2.3.1 Generator Set Induction (GSI)

Syntax

`apply gsi`[`!`] `to` <*goal-identifier*> `on` <*variable-name*> `with` <*generator-set*> `.`

where < *goal-identifier* > is the identifier of the goal on which we want to apply the **GSI** rule, < *variable-name* > is the variable on which we want to apply induction, and < *generator-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 X:Nat with 0 ;; s(Y:Nat) .
```

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

See examples of use in Sections 1.6, 1.7, 1.8.1, 1.8.2, 1.9, 1.10.1, and 3.2.

Requirements

The goal's multiclause contains the variable on which we want to apply **GSI**, and the generator set does not contain any variable already existing in the clause of goal. Furthermore, the generator set should be a correct generator set (modulo the axioms holding among constructor terms) for ground constructor terms the chosen sort. In this alpha version of the NuITP, 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.

2.3.2 Narrowing Induction (NI)

Syntax

`apply ni`[`!`] `to` <*goal-identifier*> `on` <*subterm*> `.`

where < *goal-identifier* > 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(Q:List Y:Elt) .
```

```
NuITP> apply ni! to 0.1 on rev(Q:List Y:Elt) .
```

See examples of use in Sections 3.3.

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.

2.3.3 Lemma Enrichment (LE)

Syntax

`apply le[!] to` <*goal-identifier*> `with` <*multiclause*> `.`

where <*goal-identifier*> 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 .
```

See an example of use in Section 3.4.

2.3.4 Split (SP)

Syntax

`apply sp[!] to` <*goal-identifier*> `with` <*disjunction*> `sub` <*substitution*> `.`

where <*goal-identifier*> 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 + 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) .
```

Requirements

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

2.3.5 Case (CAS)

Syntax

`apply cas[!] to` <*goal-identifier*> `on` <*variable-name*> `with` <*generator-set*> `.`

where <*goal-identifier*> is the identifier of the goal to which we want to apply the **CAS** rule, <*variable-name*> is the variable on which we want to apply cases, and <*generator-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.

Examples

Applied on variables:

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

Applied on *skolem* constants:

```
NuITP> apply cas to 0.1 on X1 with 0 ;; s(Y:Nat) .
```

```
NuITP> apply cas! to 0.1 on X1 with 0 ;; s(Y:Nat) .
```

Requirements

The goal's multiclause contains the variable on which we want to apply the **CAS** rule and the generator set does not contain any variable already existing in the clause of the goal. As already mentioned for the `gsi` command, in this alpha version of the NuITP, checking the correctness of the generator set is the user's responsibility.

2.3.6 Variable Abstraction (VA)

Syntax

`apply va`[`!`] `to` <*goal-identifier*> `on` <*term*> `.`

where <*goal-identifier*> 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 Y:NzNat * Z:NzNat .
```

```
NuITP> apply va! to 0.1 on Y:NzNat * Z:NzNat .
```

Requirements

The 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.

3 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.

3.1 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.[8]

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

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

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

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

    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[9] 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(Y:Nat)} = {Y:Nat, X:Nat ^ s(s(0))}
        -> [s(X:Nat + Y:Nat), X:Nat] = [X:Nat + X:Nat ^ s(s(0)), X:Nat * X: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
```

[9] All NuITP commands have to be written in a single line (see Section 4.1 for more info).

```
  Goal:
    (0 = Y:Nat) /\ s(Y:Nat) = X:Nat * X:Nat
      -> (X:Nat = X:Nat * X:Nat) /\ s(X:Nat + Y:Nat) = X:Nat + X:Nat * X: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` must be equal to `0` (`0 = Y: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.

3.2 Associativity of list concatenation

Consider the following functional module, which encodes an equational theory that consists of the specification of the natural numbers using the 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:
    (L:List @ (P:List @ Q:List)) = ((L:List @ P:List) @ Q: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 L:List 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 @ (P:List @ Q:List)) = ((nil @ P:List) @ Q:List)

  Goal Id: 0.2
  Skolem Ops:
    R.List
    m.Nat
  Executable Hypotheses:
    ((R @ P:List) @ Q:List) => (R @ (P:List @ Q:List))
  Non-Executable Hypotheses:
    None
  Goal:
    ((m ; R) @ (P:List @ Q:List)) = (((m ; R) @ P:List) @ Q: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 `L:List` in Goal `0` by `nil`. In Goal `0.2` the same variable has been instantiated by the term `(m ; R)`, which is itself and instance of the second term in the generator set we provided. Note also that both `n` and `R` 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:
    R.List
    m.Nat
  Executable Hypotheses:
    ((R @ P:List) @ Q:List) => (R @ (P:List @ Q:List))
  Non-Executable Hypotheses:
    None
  Goal:
    ((m ; R) @ (P:List @ Q:List)) = (((m ; R) @ P:List) @ Q:List)

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 teh **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 L:List with nil ;; (m:Nat ; R:List) .

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

  Goals 0.1 and 0.2 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 will not cover all possible values for our chosen variable will result in a faulty or incomplete proof.

3.3 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(Q:List Y:Elt) = Y:Elt rev(Q:List)
```

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

```
NuITP> apply ni to 0 on rev(Q:List Y:Elt) .
```

```
  Narrowing Induction (NI) applied to goal 0.

  Goal Id: 0.1
  Skolem Ops:
    0.1@1.Elt
    0.1@2.Elt
    0.1@3.List
  Executable Hypotheses:
    rev(0.1@3 0.1@2) => 0.1@2 rev(0.1@3)
  Non-Executable Hypotheses:
    None
  Goal:
    (0.1@2 rev(0.1@1 0.1@3)) = rev(0.1@3 0.1@2) 0.1@1

  Goal Id: 0.2
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    (0.2@2:Elt rev(0.2@1:Elt)) = rev(0.2@2:Elt) 0.2@1:Elt
```

This command basically *narrows* the term (i.e., it symbolically evaluates it with the equations defining the `rev` function), yielding the shown two goals. Goal `0.1` has now a ground clause, where fresh variables `0.1@1:Elt`, `0.1@2:Elt`, and `0.1@3: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:
    (0.2@2:Elt rev(0.2@1:Elt)) = rev(0.2@2:Elt) 0.2@1:Elt
```

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(Q:List Y: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.

3.4 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 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) = X:Nat * X:Nat -> X: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 X:Nat 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(Z:Nat)) * s(s(Z:Nat)) -> s(0) = s(s(Z: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(Z:Nat)) * s(s(Z:Nat)) -> s(0) = s(s(Z:Nat))

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(Z:Nat)) * s(s(Z:Nat)) -> s(0) = s(s(Z:Nat))
```

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(Z:Nat)) + Z:Nat) + s(Z:Nat) * s(Z: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:
    (N:Nat + M:Nat) = M:Nat + N:Nat

  Goal Id: 0.3.1.2
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    (N:Nat + M:Nat) = M:Nat + N:Nat
  Goal:
    s(0) = s(s(s(Z:Nat)) + Z:Nat) + s(Z:Nat) * s(Z: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 it **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 N:Nat + M:Nat => M:Nat + N:Nat .

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

  Goal Id: 0.3.1.2.1
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    (N:Nat + M:Nat) = M:Nat + N:Nat
  Goal:
    0 = s(s((s(Z:Nat) * s(Z:Nat)) + Z:Nat)) + Z:Nat -> false
```

Note that any ground instance of the goal's condition will be false, since the equality that states that `0 = s(s((s(Z:Nat) * s(Z:Nat)) + Z:Nat)) + Z:Nat` will never be true in our example. Also note that the subterm `s(Z:Nat) * s(Z: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 on Z:Nat with 0 ;; s(0) ;; s(s(W:Nat)) .

  Generator Set Induction with Equality Predicate Simplification (GSI!)
  applied to goal 0.3.1.2.1.

  Goals 0.3.1.2.1.1, 0.3.1.2.1.2 and 0.3.1.2.1.3 have been proved.

  Unproved goals:

  Goal Id: 0.3.1.1
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    (N:Nat + M:Nat) = M:Nat + N:Nat
```

At this point, the only remaining goal is the lemma we introduced with the **LE** rule, which needs to be proved[10] 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.

3.5 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

[10]Two alternative proofs of the commutativity of addition can be found in Section 1.8.

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:
    (X:Nat * Z':NzNat) = Y:Nat * Z':NzNat -> X:Nat = Y:Nat
```

We start proving our goal by first applying our well known **GSI!** rule, which will apply **GSI** followed by **EPS** in each of the generated goals:

```
NuITP> apply gsi! to 0 on X:Nat with 0 ;; 1 ;; 1 + X1:NzNat .

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

  Goal Id: 0.1
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    0 = Z':NzNat * Y:Nat -> 0 = Y:Nat

  Goal Id: 0.2
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    Z':NzNat = Z':NzNat * Y:Nat -> 1 = Y:Nat

  Goal Id: 0.3
  Skolem Ops:
    X1.NzNat
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    Z':NzNat = Z':NzNat * Y:Nat -> 1 = Y:Nat
    (X1 * Z':NzNat) = Z':NzNat * Y:Nat -> X1 = Y:Nat
  Goal:
    (Z':NzNat * Y:Nat) = Z':NzNat + X1 * Z':NzNat -> Y:Nat = 1 + X1
```

To prove Goal `0.1`, we apply **CAS!** on variable `Y:Nat` as follows:

```
NuITP> apply cas! to 0.1 on Y:Nat with 0 ;; Y1:NzNat .

  Case with Equality Predicate Simplification (CAS!) applied to goal 0.1.

  Goal 0.1.1 has been proved.

  Goal Id: 0.1.2
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    0 = Z':NzNat * Y1:NzNat -> 0 = Y1:NzNat
```

Then, we can use variable abstraction (**VA**), which will abstract the subterm of the clause that we provide as argument (i.e., `Z':NzNat * Y1:NzNat`) into a new, fresh variable:

```
NuITP> apply va! to 0.1.2 on Z':NzNat * Y1:NzNat .

  Variable Abstraction with Equality Predicate Simplification (VA!)
  applied to goal 0.1.2.

  Goal Id: 0.1.2.1
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    (0 = 0.1.2.1#1:NzNat) /\ 0.1.2.1#1:NzNat = Z':NzNat * Y1:NzNat -> 0 = Y1:NzNat
```

This command leaves the clause in Goal `0.1.2.1` ready to be simplified by means of the **CVUL** simplification rule[11], since the equality `0 = 0.1.2.1#1:NzNat` will never be true, which will falsify the condition proving the premise:

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

```
NuITP> apply cvul to 0.1.2.1 .

  Constructor Variant Unification Left (CVUL) applied to goal 0.1.2.1.

  Goal 0.1.2.1.1 has been proved.

  Unproved goals:

  Goal Id: 0.2
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    Z':NzNat = Z':NzNat * Y:Nat -> 1 = Y:Nat

  Goal Id: 0.3
  Skolem Ops:
    X1.NzNat
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    Z':NzNat = Z':NzNat * Y:Nat -> 1 = Y:Nat
    (X1 * Z':NzNat) = Z':NzNat * Y:Nat -> X1 = Y:Nat
  Goal:
    (Z':NzNat * Y:Nat) = Z':NzNat + X1 * Z':NzNat -> Y:Nat = 1 + X1
```

Now we try to prove Goal `0.2` in a very similar way by first applying **GSI!**:

```
NuITP> apply gsi! to 0.2 on Y:Nat with 0 ;; Y1:NzNat .

  Generator Set Induction with Equality Predicate Simplification (GSI!)
  applied to goal 0.2.

  Goal Id: 0.2.1
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    0 = Z':NzNat -> false

  Goal Id: 0.2.2
  Skolem Ops:
    Y1.NzNat
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    Z':NzNat = Y1 * Z':NzNat -> 1 = Y1
```

and then **CVUL** to Goal `0.2.1`:

```
NuITP> apply cvul to 0.2.1 .

  Constructor Variant Unification Left (CVUL) applied to goal 0.2.1.

  Goal 0.2.1.1 has been proved.
  Unproved goals:

  Goal Id: 0.2.2
  Skolem Ops:
    Y1.NzNat
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    Z':NzNat = Y1 * Z':NzNat -> 1 = Y1

  Goal Id: 0.3
  Skolem Ops:
    X1.NzNat
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    Z':NzNat = Z':NzNat * Y:Nat -> 1 = Y:Nat
    (X1 * Z':NzNat) = Z':NzNat * Y:Nat -> X1 = Y:Nat
  Goal:
    (Z':NzNat * Y:Nat) = Z':NzNat + X1 * Z':NzNat -> Y:Nat = 1 + X1
```

and **CAS!** to Goal `0.2.2` but, this time, on the *skolem* constant `Y1`:

```
NuITP> apply cas! to 0.2.2 on Y1 with 1 ;; 1 + Y1':NzNat .

  Case with Equality Predicate Simplification (CAS!) applied to goal 0.2.2.

  Goals 0.2.2.1 and 0.2.2.2 have been proved.

  Unproved goals:

  Goal Id: 0.3
  Skolem Ops:
    X1.NzNat
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    Z':NzNat = Z':NzNat * Y:Nat -> 1 = Y:Nat
    (X1 * Z':NzNat) = Z':NzNat * Y:Nat -> X1 = Y:Nat
  Goal:
    (Z':NzNat * Y:Nat) = Z':NzNat + X1 * Z':NzNat -> Y:Nat = 1 + X1
```

Finally, we start simplifying our remaining Goal `0.3` by first applying **CAS!**:

```
NuITP> apply cas! to 0.3 on Y:Nat with 0 ;; 1 ;; 1 + Y1:NzNat .

  Case with Equality Predicate Simplification (CAS!) applied to goal 0.3.

  Goals 0.3.1 and 0.3.2 have been proved.

  Goal Id: 0.3.3
  Skolem Ops:
    X1.NzNat
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    Z':NzNat = Z':NzNat * Y:Nat -> 1 = Y:Nat
    (X1 * Z':NzNat) = Z':NzNat * Y:Nat -> X1 = Y:Nat
  Goal:
    (X1 * Z':NzNat) = Z':NzNat * Y1:NzNat -> X1 = Y1: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 0.3.3 .

  Clause Subsumption (CS) applied to goal 0.3.3.

  Goal 0.3.3.1 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.

3.6 Reversing Palindromes

Consider the following conditional equational theory, which extends the theory of Section 3.3 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(L:List) -> L:List = rev(L: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 3.3:

```
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(Q:List Y:Elt) = Y:Elt rev(Q:List)

  Goal Id: 0.2
  Skolem Ops:
    None
  Executable Hypotheses:
    rev(Q:List Y:Elt) => Y:Elt rev(Q:List)
  Non-Executable Hypotheses:
    None
  Goal:
    true = pal(L:List) -> L:List = rev(L: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(L:List) .

  Narrowing Induction with Equality Predicate Simplification (NI!)
  applied to goal 0.2.

  Goals 0.2.1 and 0.2.2 have been proved.

  Goal Id: 0.2.3
  Skolem Ops:
    0.2.3@1.Elt
    0.2.3@2.List
  Executable Hypotheses:
    rev(Q:List Y:Elt) => Y:Elt rev(Q:List)
    pal(0.2.3@2) = true -> rev(0.2.3@2) => 0.2.3@2
  Non-Executable Hypotheses:
    None
  Goal:
    true = pal(0.2.3@2) -> 0.2.3@2 = rev(0.2.3@2)
```

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 0.2.3 .

  Clause Subsumption (CS) applied to goal 0.2.3.

  Goal 0.2.3.1 has been proved.
```

```
Unproved goals:

Goal Id: 0.1
Skolem Ops:
  None
Executable Hypotheses:
  None
Non-Executable Hypotheses:
  None
Goal:
  rev(Q:List Y:Elt) = Y:Elt rev(Q:List)
```

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.

4 Troubleshooting

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

4.1 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.
```

4.2 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, 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.

4.3 Writing and running scripts

As pointed above, NuITP relies on Maude's I/O capabilities to access the file system, which in turn wrap the C *stdio* library (see Chapter 9 of [2]). 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 12a version, NuITP saves scripts (and also exports proof reports) 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* file name 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] 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.

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

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

NuITP Reference Sheet

Quit and Help	
`help`	Shows the help message
`quit`	Exits NuITP
`q`	Exits NuITP (short version)

Scripts and Report	
`load FILE`	Loads a script located in `FILE`
`save FILE`	Saves the current session script in `FILE`
`export FILE`	Creates a LaTeX report of the current session in `FILE`

Showing Data	
`show log .`	Shows the full session log
`show module .`	Shows the active module
`show goals .`	Shows all the goals of the proof
`show frontier .`	Shows the open goals of the proof
`show goal GID .`	Shows the goal with `GID` identifier

Set and Undo	
`set module MODNAME .`	Sets module `MODNAME` as the active module
`set goal GOAL .`	Sets the initial goal (resets the proof)
`undo GID .`	Undoes any rule applied to the goal `GID`

Simplification Rules	
`apply eps to GID .`	Applies Equality Predicate Simplification to the goal `GID`
`apply cvul to GID .`	Applies Constructor Variant Unification Left to the goal `GID`
`apply cvufr to GID .`	Applies Constructor Variant Unification Failure Right to the goal `GID`
`apply subl to GID .`	Applies Substitution Left to the goal `GID`
`apply subr to GID .`	Applies Substitution Right to the goal `GID`
`apply cs to GID .`	Applies Clause Subsumption to the goal `GID`
`apply varsat to GID .`	Applies Variant Satisfiability to the goal `GID`
`apply eq to GID with EQ .`	Applies Equality to the goal `GID` using the **oriented** hypothesis `HYP`
`apply eq to GID with HYP sub SUB .`	Applies Equality to the goal `GID` using the **oriented** hypothesis `HYP` and substitution `SUB`

Induction Rules	
`apply gsi to GID on VAR with GENSET .`	Applies Generator Set Induction to the goal `GID`, on variable `VAR` with generator set `GENSET`
`apply ni to GID on TERM .`	Applies Narrowing Induction to the goal `GID` on the (sub)term `TERM`
`apply le to GID with LEMMA .`	Applies Lemma Enrichment to the goal `GID` with lemma `LEMMA`
`apply sp to GID with DIS sub SUB .`	Applies Split to the goal `GID` with disjunction `DIS` and substitution `SUB`
`apply cas to GID on VAR with GENSET .`	Applies Case to the goal `GID` on variable `VAR` with generator set `GENSET`
`apply va to GID on TERM .`	Applies Variable Abstraction to the goal `GID` on (sub)term `TERM`

Troubleshooting	
I don't know how to start NuITP	`$ maude -allow-files NuITP.maude`
I can't load or save scripts	Have you started Maude with the –allow-files flag?
I can't set my module	Did you load your module in Maude before starting NuITP?
Yes but I still can't set my module!	Check twice your module's name. No quote at the beginning is needed.
Why I always get "Error parsing command"?	Did you forget the ending dot? Check if the command requires it. Also, try to use more parenthesis when specifying the arguments in the apply rules.
Why I always get a failure message when trying to apply a rule to a goal that is in the frontier?	Most of the rules have theory requirements that, if not met, will prevent them to be applied. Check the manual to learn more about them.
Is there something I need to do before starting a proof session?	Yes, check that your theory has the proper `ctor`, `variant`, and `metadata` attributes set.
What is Σ_1? And Ω?	Check Section 1.3 of the manual.
What's the difference, for example, the `ni` rule and the `ni!` rule?	Induction rules, as well as the `EQ` rule, have extended (`!`) versions that apply `EPS` automatically to their computed subgoals in order to simplify them.