Chapter 8

Rules and Tactics

Logically, all formal reasoning in Nuprl is based on the basic inference rules of *intuitionistic type theory* [ML84, CAB+86]. In practice, however, reasoning with basic inference rules can become very tedious. Therefore Nuprl provides the concepts of proof *tactics*, i.e. programmed applications of inference rules. Proof tactics range from straightforward applications of inference rules to fully automated proof search mechanisms for certain domains. Most tactics are deal with intellectually trivial, but formally lengthy proof details, enabling the user to concentrate on the more interesting aspects of a proof. Others mimic the particular style of reasoning in a certain application domain and can only be applied in this context. Technically, all refinement steps in Nuprl are executions of tactics. Basic inference rules, although explicitly present in the Nuprl library, cannot be invoked directly but only through conversion into tactics.

In the rest of this chapter we will describe the structure of basic inference rules and tactics as well as the most important groups of tactics that are currently implemented. Nuprl's type theory, its semantics, and the guiding principles for its development is explained in [CAB+86]. It should be noted, however, that Nuprl's type theory is *open-ended* and that fundamentally new concepts and their inference rules are added whenever this is turns out to be necessary.

A complete list of the current set of inference rules can be found in Appendix A.3.

8.1 Rules

Inference rules characterize the semantics of all formal expressions in Nuprl. For each type construct they describe how to form types and the conditions for two types to be equal, how to form members of types and the prerequisites for two members of a type to be equal in that type.

Nuprl's proof calculus is based on the notion of *sequents*. These are objects of the form $x_1{:}T_1, \ldots, x_n{:}T_n \vdash C$, which should be read as "Under the *hypotheses* that the x_i are variables of type T_i a member of the *conclusion* C can be constructed" (see Section 6.1.1 for details).

In Nuprl a proof for such a sequent is developed in a *top-down* fashion. Proof rules *refine* a goal sequent obtaining subgoal sequents whose proofs would suffice to validate the original goal. They are described by rule schemata with placeholders for lists of hypotheses and conclusions. A rule

$$\begin{array}{l}
\Gamma \vdash C \quad \lfloor \textsf{ext}\ m \rfloor \\
\quad \textbf{by}\ \textit{rule-name (\& optional arguments)} \\
\quad \Gamma_1 \vdash C_1 \quad \lfloor \textsf{ext}\ m_1 \rfloor \\
\qquad\qquad \ldots \\
\quad \Gamma_n \vdash C_n \quad \lfloor \textsf{ext}\ m_n \rfloor
\end{array}$$

expresses the fact that the goal sequent $\Gamma \vdash C$ is provable if all the subgoals $\Gamma_i \vdash C_i$ are. Furthermore, it also describes how to construct an (implicitly present) member m of C from members m_i of the subgoal conclusions C_i.

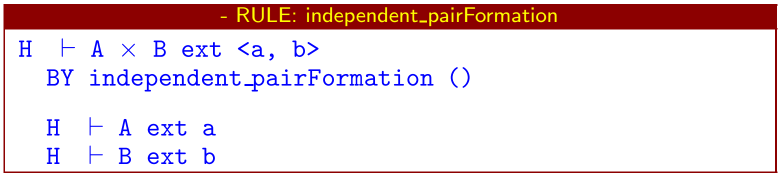
Rules may operate both on the conclusion or the hypotheses of a proof goal. In most cases they simply decompose a type or a member of a type into smaller components. The rule for the formation of pairs, for instance, states that in order to form a member $\langle s,t\rangle$ of a product type $S{\times}T$ it suffices to form a member s of the type S and a member t of T.

$$\begin{array}{l} \Gamma \vdash S{\times}T \quad \lfloor\textbf{ext}\ \langle s,t\rangle\rfloor \\ \quad \textbf{by}\ \texttt{independent\_pairFormation} \\ \quad \Gamma \vdash S \quad \lfloor\textbf{ext}\ s\rfloor \\ \quad \Gamma \vdash T \quad \lfloor\textbf{ext}\ t\rfloor \end{array}$$

This rule can be applied to any goal whose conclusion is a product type. In this case S is matched against the left, and T against the right component of that product, while Γ is matched against the complete list of hypotheses. As a result, NUPRL generates two new proof goals, the first with the left component as proof goal and the second one with the right component. The list of hypotheses remains unchanged in both cases.

8.1.1 Representation of Inference Rules

NUPRL's inference rules are not hard-wired into the code of the system but explicitly represented in the library as objects of kind *rule*. Rule definitions are 'terms' in the sense described in Chapter 5 and they can be edited like any other term. The rule for the formation of pairs, for instance, is represented by an object called `independent_pairFormation`.



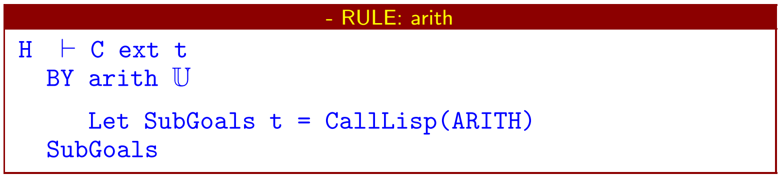
- RULE: independent_pairFormation

```
H  ⊢ A × B ext <a, b>
  BY independent_pairFormation ()

  H  ⊢ A ext a
  H  ⊢ B ext b
```

The fact that the rule object is almost identical to the rule described on paper, makes it very easy to verify the implementation of intuitionistic type theory in NUPRL. The explicit representation of inference rules in NUPRL also allows a user to modify the logic represented in system by adding new rules or deactivating existing ones. Thus NUPRL supports any logic that can be represented as a top-down sequent calculus.

NUPRL provides the basic mechanisms for applying rule objects to proofs. In most cases this means matching the rule's top goal against the current proof goal and extending the proof tree by the instantiated sub-goals of the rule. Some rules, such as the decision procedure `arith` provide explicit calls to special purpose algorithms that will be executed when NUPRL applies the rules.



- RULE: arith

```
H  ⊢ C ext t
  BY arith 𝕌

     Let SubGoals t = CallLisp(ARITH)
  SubGoals
```

8.1.2 Rule Arguments

In most cases, the application of an inference rule to a proof goal requires more information than just the name of the rule. For instance, if a rule shall be applied to one of the hypotheses, it is necessary to identify this hypothesis; or if a rule creates new variables in the subgoals, it is necessary to give names to these variables. Because inference rules are supposed to be applied schematically, this information will not be determined automatically but has to provided as additional arguments. Inference rules may require the following arguments.

Hypothesis index. If a rule shall be applied to a particular hypothesis of the proof goal, the corresponding index i of that hypothesis in the hypotheses list of the goal must be provided. The rule `hypothesis`, for instance, can be used to prove a goal if the conclusion is identical (α-equal) to one of the assumptions, but the index of that assumption must be given.

$\Gamma, x{:}T, \Delta \vdash T$ $\lfloor$**ext** $x\rfloor$
  by `hypothesis` $\boxed{i}$

```
- RULE: hypothesis
H x:A, J ⊢ A ext x
  BY hypothesis #$i
  No Subgoals
```

Variable names. Many inferences require the creation of new variables when refining a proof goal. The corresponding proof rules do not choose the names for these variables automatically but expect them to be provided explicitly. The rule `functionEquality`, for instance, is used to prove two dependent function types $x_1{:}S_1{\to}T_1$ and $x_2{:}S_2{\to}T_2$ equal. It creates two subgoals: the two domains S_1 and S_2 must be equal, and the two ranges $T_1[x/x_1]$ and $T_2[x/x_2]$ must be equal for each argument $x \in S_1$. The second subgoal contains a new variable, whose name must be given.

$\Gamma \vdash x_1{:}S_1{\to}T_1 = x_2{:}S_2{\to}T_2 \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
  by `functionEquality` $\boxed{x}$
    $\Gamma \vdash S_1 = S_2 \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$
    $\Gamma, x{:}S_1 \vdash T_1[x/x_1] = T_2[x/x_2] \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$

```
- RULE: functionEquality
H  ⊢ (x1:a1 → b1) = (x2:a2 → b2)
  BY functionEquality y

  H  ⊢ a1 = a2
  H y:a1 ⊢ !subst(b1; x1.y) = !subst(b2; x2.y)
```

Universe level. Because of the expressiveness of type theory, the well-formedness of a type expression cannot be decided automatically, but must be established in the course of a proof development. Whenever a rule creates a new declaration in one of the its subgoals, the well-formedness of the corresponding type must be established. If this cannot not guaranteed by the proof of the other subgoals (as in the case of `functionEquality` it must be proven as a separate subgoal. The rule `lambdaEquality`, for instance, proves the equality of two λ-terms in a function type $x{:}S{\to}T$. For this purpose, it declares a new variable X' of type S and adds a subgoal stating that S belongs to some universe $\mathbb{U}_j$. The level j of that universe must be given.

$\Gamma \vdash \lambda x_1.t_1 = \lambda x_2.t_2 \in x{:}S{\to}T$ $\lfloor$Ax$\rfloor$
  by `lambdaEquality` $\boxed{j}$ x'
    $\Gamma, x'{:}S \vdash t_1[x'/x_1] = t_2[x'/x_2] \in T[x'/x]$ $\lfloor$Ax$\rfloor$
    $\Gamma \vdash S \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$

```
- RULE: lambdaEquality
H  ⊢ (λx1.b1) = (λx2.b2)
  BY lambdaEquality !parameter{i:l} z ()

  H z:A ⊢ !subst(b1; x1.z) = !subst(b2; x2.z)
  H  ⊢ A = A
```

Values for variables. Some proof goals can only be decomposed if it is known how to instantiate a certain variable. To prove $\exists x{:}S.T[x]$ – which is the same as the dependent product $x{:}S{\times}T[x]$), for instance, one has to provide a value s for the existentially quantified variable x and can then go on to prove $T[s]$. The corresponding rule `dependent_pairFormation` requires this term as one of its arguments.

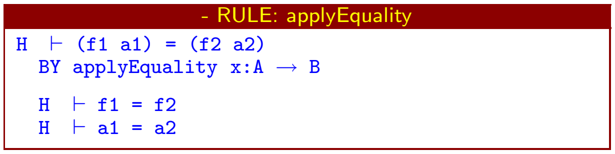
$\Gamma \vdash x{:}S{\times}T$ $\lfloor$**ext** $\langle s, t\rangle\rfloor$
  by `dependent_pairFormation` j $\boxed{s}$ x'
    $\Gamma \vdash s \in S$ $\lfloor$Ax$\rfloor$
    $\Gamma \vdash T[s/x]$ $\lfloor$**ext** $t\rfloor$
    $\Gamma, x'{:}S \vdash T[x'/x] \in \mathbb{U}_j$ $\lfloor$Ax$\rfloor$

```
- RULE: dependent_pairFormation
H  ⊢ x:A × B ext <a, b>
  BY dependent_pairFormation !parameter{i:l} a y ()

  H  ⊢ a = a
  H  ⊢ !subst(B; x.a) ext b
  H y:A ⊢ !subst(B; x.y) = !subst(B; x.y)
```

Type of a subterm. Sometimes decomposing a proof goal into smaller components involves proving that certain subterms belong to a type that cannot immediately constructed from the types mentioned in the goal. Proving the equality of two function applications $f_1 t_1$ and $f_2 t_2$ in a type T, for instance, requires proving f_1 and f_2 equal in some type $x{:}S{\to}T$. As this type cannot be derived from T it must be given explicitly.

$\Gamma \vdash f_1 t_1 = f_2 t_2 \in T[t_1/x]$ ⌊Ax⌋
  by `applyEquality` $\boxed{x{:}S{\to}T}$
    $\Gamma \vdash f_1 = f_2 \in x{:}S{\to}T$ ⌊Ax⌋
    $\Gamma \vdash t_1 = t_2 \in S$ ⌊Ax⌋



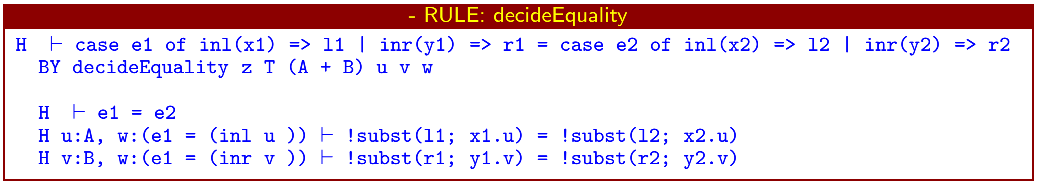
- RULE: applyEquality

```
H  ⊢ (f1 a1) = (f2 a2)
  BY applyEquality x:A → B

  H  ⊢ f1 = f2
  H  ⊢ a1 = a2
```

Term dependency. Applying a proof rule may sometimes involve replacing an sub-expression e in the goal by some other expression. If e occurs several times in the goal, one must indicate which of the occurrences of e shall be replaced and which one shall not. Technically, this is done by providing a term substitution: all occurrences of e in the term C that shall be affect by the rule application are replaced by a new variable z. The rule will then substitute every occurrence of z by the new expression. Both z and the modified term $C[z]$ must be given as arguments. As an example, consider the rule `decideEquality` for proving two case analyses equal.

$\Gamma \vdash \text{case } e_1 \text{ of inl}(x_1) \mapsto u_1 \mid \text{inr}(y_1) \mapsto v_1 = \text{case } e_2 \text{ of inl}(x_2) \mapsto u_2 \mid \text{inr}(y_2) \mapsto v_2 \in C[e_1/z]$ ⌊Ax⌋
  by `decideEquality` $\boxed{z\ C}$ $S{+}T$ s t y
    $\Gamma \vdash e_1 = e_2 \in S{+}T$ ⌊Ax⌋
    $\Gamma,\ s{:}S,\ y{:}\ e_1{=}\text{inl}(s) \in S{+}T \vdash u_1[s/x_1] = u_2[s/x_2] \in C[\text{inl}(s)/z]$ ⌊Ax⌋
    $\Gamma,\ t{:}T,\ y{:}\ e_1{=}\text{inr}(t) \in S{+}T \vdash v_1[t/y_1] = v_2[t/y_2] \in C[\text{inr}(t)/z]$ ⌊Ax⌋



- RULE: decideEquality

```
H  ⊢ case e1 of inl(x1) => l1 | inr(y1) => r1 = case e2 of inl(x2) => l2 | inr(y2) => r2
  BY decideEquality z T (A + B) u v w

  H  ⊢ e1 = e2
  H u:A, w:(e1 = (inl u )) ⊢ !subst(l1; x1.u) = !subst(l2; x2.u)
  H v:B, w:(e1 = (inr v )) ⊢ !subst(r1; y1.v) = !subst(r2; y2.v)
```

In most cases, the arguments of an inference rule can easily be determined from the proof goal. Nuprl's tactic collection therefore provides a set of single-step inference tactics that try to compute the arguments required by the corresponding inference rule from the actual proof context before executing the rule (see Section 8.1.3 below). In some situations, however, the relation between the proof goal and the arguments of inference rule is not so obvious and it is necessary that the user provides the rule arguments explicitly in order to be able to complete the proof.

8.1.3 Converting rules into tactics

Nuprl's basic mechanism for creating and modifying proofs is the application of proof *tactics*. These are functions that take a sequent (i.e. the goal) and generate a list of sequents (i.e. the subgoals) as well as a *validation*, which shows that a proof for the main goal can be constructed from proofs for all the subgoals.

Basic inference rules cannot be applied directly to a proof goal, but first have to be converted into a tactic. Technically, this is done by calling a meta-level function `refine` that takes a primitive rule, that is the name of a rule object and the corresponding rule arguments, and generates a tactic that executes the rule. The function `refine` encodes Nuprl's mechanism for applying rule objects to proofs (c.f. Section 8.1.1 above) and is the only method for creating proof tactics from scratch.

This guarantees that all proof steps are eventually based on primitive inference rules and thus correct with respect to the implemented proof calculus.

In most cases, the applicable inference rule and its arguments can easily be determined from the proof context. Nuprl's library therefore contains a small collection of single-step inference tactics that subsumes the complete set of elementary inference rules. Almost all primitive inferences can be expressed the single-step decomposition tactics `D`, `MemCD`, `EqCD`, `MemHD`, `EqHD`, `MemTypeCD`, `EqTypeCD`, `MemTypeHD`, `EqTypeHD`, and `NthHyp`. These tactics, which are described in detail in Section 8.3.1, uniformly apply to both the hypotheses and the conclusion of a proof goal. Often a user only has to give the index of the *goal clause* to which they shall be applied. The tactic then analyzes the syntactical structure of the indicated clause, identifies the applicable rule, and tries to determine its arguments from the proof context.

For the sake of efficient interaction, the heuristics built into these tactics only determine variable names, universe levels, types of subterms, and term dependencies, while the user always has to provide the clause index and values to be substituted for variables.[1] A user may also want to override the choices made by the single-step decomposition tactics. For this purpose Nuprl provides a collection of *tacticals* (i.e. functions that take tactics as arguments and generate tactics) that enable a user to supply rule arguments explicitly (see Section 8.2.2 for details). Arguments may be supplied to tactics as follows.

Clause index. Most single-step tactics require a clause index to be given as explicit argument. By convention (see Section 8.2.1.1 for a discussion) the index `0` stands for the conclusion of the proof goal, while positive numbers indicate hypotheses. As a convenience, negative indices can be used to count backwards from the end of the hypothesis list. The index `(-1)` indicates the last hypothesis in the goal, `(-2)` the second to last, etc.

The rule `independent_pairFormation`, for instance, operates on the conclusion and is represented by the single-step tactic `D 0`. The rule `hypothesis` i is represented by `NthHyp` i.

Variable names. All tactics in Nuprl's library automatically assign names to new variables. In rare cases a user may want to select different names. The tactical `New` takes as an input a list of variables and a tactic and generates a tactic that uses the provided variables instead of the automatically chosen ones.

The the tactic `D 0`, which usually suffices to represent the rule `functionEquality` x, can be forced to choose the name x for the new variable by writing `New` [x] (D 0).

Universe level. Universe levels can usually be determined from the immediate proof context and the well-formedness theorems of user-defined concepts. In some cases, the heuristics choose a rather arbitrary, high universe level. To enforce a particular level, one may use the tactical `At`.

The the tactic `D 0`, which usually suffices to represent the rule `lambdaEquality` j x', can be forced to choose the universe level j by writing `At` $\mathbb{U}_j$ `(D 0)`. In addition, one could also enforce the use of x' as new variable name as in `At` $\mathbb{U}_j$ `(New` [x']`(D 0))`

Values for variables. The `With` tactical can be used to supply terms that are needed for the instantiation of variables. The rule `dependent_pairFormation` j s x', for instance, requires the term s to be provided explicitly. It is represented by the tactic `With` s `(D 0)`. A universe level and a variable name may also be provided by writing `At` $\mathbb{U}_j$ `(With` s `(New` [x'] `(D 0)))`.

[1] Nuprl's library contains proof tactics that attempt to determine clause index and values for variables automatically. But since the underlying heuristics are often time consuming and sometimes choose values leading to subgoals that cannot be proven, they are not suited for representing single-step inferences.

Type of a subterm Although type checking in general is undecidable for intuitionistic type theory, the types of subterms occurring in a proof goal can almost always be determined automatically. A user may override the heuristically chosen type if it is too coarse for completing the proof or provide a type explicitly to improve the efficiency of proof checking. The tactical `With` can be used for this purpose as well.

The tactic MemCD, which usually suffices to represent the rule `applyEquality` $x{:}S{\rightarrow}T$, can be forced to use a particular function type $x{:}S{\rightarrow}T$ by writing `With` $x{:}S{\rightarrow}T$ `MemCD`.

Term dependency. Term dependencies can be supplied to a tactic with the `Using` tactical. The tactic `D 0`, which usually suffices to represent the rule `decideEquality` $z\ C\ S{+}T\ s\ t\ y$, can be forced to use a particular dependency $C[z]$ with a new variable z by writing `Using [`z`,`C`] (D 0)`. Note that a disjunctive type and up to three new variables could be provided as well, as in `Using [`z`,`C`] (With` $S{+}T$ `(New [`s`;`t`;`y`] (D 0)))`.

Rule selection. In some rare cases it is impossible to determine the exact rule that shall be applied to a particular proof clause. For instance, there are two different rules can be applied to a disjoint union $S{+}T$ in the conclusion of a goal.

$$\begin{array}{l} \Gamma \vdash S{+}T \quad \lfloor\textbf{ext } \mathsf{inl}(s)\rfloor \\ \quad \textbf{by } \texttt{inlFormation } j \\ \quad \Gamma \vdash S \quad \lfloor\textbf{ext } s\rfloor \\ \quad \Gamma \vdash T \in \mathbb{U}_j \quad \lfloor\mathsf{Ax}\rfloor \end{array} \qquad \begin{array}{l} \Gamma \vdash S{+}T \quad \lfloor\textbf{ext } \mathsf{inr}(t)\rfloor \\ \quad \textbf{by } \texttt{inrFormation } j \\ \quad \Gamma \vdash T \quad \lfloor\textbf{ext } t\rfloor \\ \quad \Gamma \vdash S \in \mathbb{U}_j \quad \lfloor\mathsf{Ax}\rfloor \end{array}$$

There is no simple heuristic for determining from the proof context which of the two must be applied (although this is possible in limited application domains). Instead, a user has to *select* between the two rules, using the tactical `Sel`. The rule `inlFormation` j will be executed by writing `Sel 1 (D 0)`, while `Sel 2 (D 0)` leads to the execution of `inrFormation` j. In both cases a universe level can also be supplied.

A complete description of all the inference rules currently present in the NUPRL system as well as the corresponding single-step tactics can be found in Appendix A.3.

8.2 Introduction to Tactics

The creation and modification of proofs in NUPRL is based on the concept of *tactics*. Logically, tactics are functions that represent valid refinements. They convert a proof goal into a list of subgoals such that the validity of all the subgoals implies the validity of the initial proof goal. They range from elementary inference steps to sophisticated proof strategies that can solve major proof problems automatically.

Technically, tactics are functional programs written in the ML programming language (see Appendix B). They take as input a proof goal and return a list of proof goals and a *validation*. The validation is a function that constructs a proof for the initial goal from proofs for the subgoals and thus validates the refinement step performed by the tactic.

Tactics are usually initiated from within the proof editor. After the user types in the tactic the proof editor applies it to the current proof goal. The proof goals generated by the tactic will be added to the proof tree as *children* of the current proof node and displayed as subgoal sequents still to be proven. The validation is stored in the proof node as well but remains invisible. Upon completion of the proof the validations of all proof nodes are composed into a proof for the initial proof goal, which provides computational evidence for the validity of the initial theorem.

Tactics can either be built from elementary inference rules using the function `refine` (see Section 8.1.3), or by composing existing tactics into new ones using tacticals (see Section 8.8) or more sophisticated ML programs that analyze the proof context before initiating a refinement step. This makes sure that all refinements performed by tactics are eventually based on elementary inference rules and thus correct with respect to the underlying logic.[2] Applying a tactic to a proof goal may produce incomplete proofs or not terminate, but it *always results in a valid proof*.

Nuprl's standard library contains a large collection of useful tactics that have been developed over the past 15 years. Users may extend that collection by adding their own tactics as code-objects to their personal directories. But in most cases it is sufficient to use the existing tactics and to combine them through tacticals.

In the rest of this chapter we will describe the most important standard tactics contained in the library. Further tactics may be found by inspecting Nuprl's library of standard theories.

8.2.1 Tactic Arguments

Invoking a tactic often requires certain arguments to be supplied. These may be indices of hypotheses to which the tactic shall be applied (type `int`), Nuprl terms that instantiate variables (type `term`), new variables to be generated (type `var`), universes to be used in well-formedness goals (type `term`), names of library objects to be consulted (type `token`), substitutions to be applied (type `(var#term) list`), sub-tactics to be used during refinement (type `tactic`), and others.

The proof goal (type `proof`) to which the tactic shall be applied is always the last argument of a tactic. If the tactic is invoked in the proof editor, the user does not have to supply this argument, as the proof editor will automatically insert it.

Unless otherwise stated, we assume that arguments to tactics have the following types and uses:

$T*$	: `tactic`	
$c*$	: `int`	*clause index.*
$i*$	: `int`	*hypothesis index.*
$t*$	: `term`	*term of* Nuprl*'s type theory*
$n*$	: `tok`	*name of lemma object in* Nuprl*'s library*
$a*$	: `tok`	*name of abstraction object in* Nuprl*'s library*
$v*$	: `var`	*variables in terms of* Nuprl*'s object language*
$l*$	: `tok`	*subgoal label*
$p*$	: `proof`	*current proof goal*

A suffix s on the name of an argument indicates that it is a list. For example vs is considered to have type `var list`.

8.2.1.1 Referring to hypotheses in a sequent

Hypotheses are conventionally numbered from left to right, starting from 1. These hypothesis numbers are displayed by the proof editor, and tactics usually refer to hypotheses by these numbers. Sometimes, it is convenient to consider the hypotheses numbered from right to left, and for this reason tactics consider a hypotheses list $H_1, \ldots, H_n$ to also be numbered $H_{-n}, \ldots, H_{-1}$. Occasionally, the index n+1 or 0 is used to refer to the position to the right of the last hypothesis.

[2] An experienced Nuprl hacker will, of course, find ways to bypass these mechanisms and modify proofs without using elementary inference rules. However, the dependency tracking mechanisms of Nuprl's library are able to detect theorems whose proofs were constructed that way and to identify all library objects that depend on such theorems. Thus it is possible to account for the validity of theorems even in the presence of hacks.

There are tactics which work in similar ways on both hypotheses and the conclusion. In this case, we call the hypothesis and conclusion collectively *clauses*, refer to the conclusion as *clause 0*, and to hypothesis i ($i{\neq}0$) as *clause i*.

As a convention in this manual, we prefix a hypothesis with a number followed by a period if we want to indicate explicitly the number of a hypothesis in a schematic sequent. For example, if hypothesis i is proposition P, we write the hypothesis as $i.\,P$.

8.2.1.2 Universes and Level Expressions

In Nuprl's type theory, types are grouped together into universes. Types built from the base types such as $\mathbb{Z}$ or `Atom` using the various type constructors are in universe $\mathbb{U}_1$. The subscript 1 is the *level* of the universe. Types built from universe terms with level at most i are in universe $\mathbb{U}_{i+1}$. Universe membership is cumulative; each universe also includes all the types in lower universes.

Since propositions are encoded as types, propositions reside in universes too. In keeping with the propositions-as-types encoding, we define a family of propositional universe abstractions $\mathbb{P}_1$, $\mathbb{P}_2$, ..., which unfold to the corresponding primitive type universe terms $\mathbb{U}_1$, $\mathbb{U}_2$, ...

If one is only allowed to use constant levels for universes, one often has to choose arbitrarily levels for theorems. One would then find that one needed theorems that were stated at a higher level, and would have to reprove those theorems. This was the case in Nuprl 3 and earlier releases.

Nuprl now allows one to prove theorems that are implicitly quantified over universe levels. Quantification is achieved by parameterizing universe terms by *level expressions* rather than natural number constants. The syntax of level expressions is given by the grammar:

$$L \;::=\; v \;\mid\; k \;\mid\; L\ i \;\mid\; L' \;\mid\; [L\,|\cdots|\,L]$$

The v are level-expression variables, which can be arbitrary alphanumeric strings. They are implicitly quantified over all positive integer levels. The k are level expression constants, which can be arbitrary positive integers. The i are level expression increments and must be non-negative integers. The expression $L\ i$ is interpreted as standing for levels $L{+}i$. L' is an abbreviation for L `1`. The expression $[L_1\,|\cdots|\,L_n]$ is interpreted as being the maximum of the expressions $L_1 \cdots L_n$.

Usually when stating theorems, only level expressions of the form v and v' need be used. Other expressions get automatically created by tactics. Further, it is sufficient to use a single level-expression variable throughout a theorem statement, as two occurrences of the same level-expression variable will not be related. For example, we normally prove the theorem $\forall A, B{:}\mathbb{P}_i.\ A \Rightarrow (B \Rightarrow A)$ rather than $\forall A{:}\mathbb{P}_i.\forall B{:}\mathbb{P}_j.\ A \Rightarrow (B \Rightarrow A)$.

8.2.2 Optional Arguments

Unlike Lisp functions, ML functions cannot take optional arguments, although it is natural to want to write tactics which do take optional arguments. One approach is to provide a set of variants of each tactic for the most common combinations of arguments. This can be confusing, and places an extra burden on the user who has to keep track of these variants.

Nuprl allows optional arguments to be passed to tactics by providing tacticals (see Section 8.8 below) that attach these arguments to the proof argument (i.e. the last argument) of a tactic (c.f. Section 8.2.1). Currently, this is supported for arguments of type `int`, `tactic`, `term`, `tok`, `var` and `(var#term) list`. Each argument is also given a token label, and arguments are looked up by these labels. Sets of arguments are maintained on a stack, which enables nesting of tactics that use optional arguments. Tacticals for manipulating these arguments are:

`New` `(`$[v_1;\ldots;v_n]$ `:var list)` T

Runs tactic T with variables v_1 to v_n as optional arguments, Typically, `New` is used to supply a tactic with names for newly created variables. The argument labels of the v_i are `‘v1‘` to `‘vn‘`.

`With` (t `:term`) T

Runs tactic T with term t as optional argument, which may, for instance, be a term to be instantiated for a variable or the type of some subgoal. The argument label of t will be `‘t1‘`.

`At` (U `:term`) T

Runs tactic T with term U as a optional `‘universe‘` argument. U should either be either a type universe or a propositional universe term.

`Using` (*sub* `:(var#term) list`) T

Runs tactic T with the substitution *sub* as optional `‘sub‘` argument. The substitution *sub* may be applied to instantiate variables or to indicate dependencies in some term.

`Sel` (k `:int`) T

Runs tactic T with the integer k as optional argument. `Sel` is used for selecting a simple component of a formula or a subterm of a term. The argument label of k will be `‘n‘`

`Thinning` T

Runs tactic T with the token `‘yes‘` as optional `‘thinning‘` argument. Some tactics may optionally remove (or *thin*) hypotheses that are considered superfluous after a refinement step. `Thinning` causes T's default behavior to be to thin.

`NotThinning` T

Runs tactic T with the token `‘no‘` as optional `‘thinning‘` argument. This causes T's default behavior to be to not thin.

`WithArgs` (*args* `:(tok#arg) list`) T

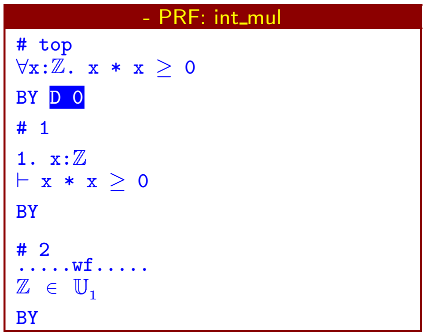
Run tactic T with the arguments in *args* on the top of the stack. `arg` is an ML abstract data type, defined as the disjoint union of the types `int`, `tactic`, `term`, `tok`, `var` and `(var#term) list`. There are injection and projection functions for each of these types, such as `int_to_arg` and `arg_to_int`. All the above tacticals are special cases of `WithArgs`

Each tactic description in this chapter includes information on the optional arguments (if any) that it takes. Note that some tactics do useful preprocessing on some of their arguments. In these cases there would be a performance penalty if such arguments were supplied.

8.2.3 Proof Annotations

Nuprl *proof* terms can be annotated with extra information that is not relevant to the logical correctness of a proof but may assist in structuring proofs and identifying applicable tactics.

Goal Labels. Nuprl tactics generate various kinds of subgoals. Some express the main proof idea, while others are only auxiliary or well-formedness goals. In inductive proofs one distinguishes the base case from the step cases. Usually, different kinds of goals have to be treated differently, so one would like subsequent tactics to discriminate on subgoal kind. Sometimes a subgoal's kind can be deduced directly from its structure, but this can be a error-prone process. Thus the tactics that generate the subgoals often attach explicit labels to them indicating their kind. Labels consist of an ML token and an optional number. Typical examples of labels are `main`, `upcase`, `aux`, and `wf`. Sometimes tactics generate a set of subgoals of the same kind, where the order of the subgoals is important. The optional numbers are used to discriminate between these subgoals. Most descriptions of tactics include information on subgoal labeling. It is also a simple matter to find out what labels are generated by experimentation.



```
- PRF: int_mul
# top
∀x:ℤ. x * x ≥ 0
BY D 0
# 1
1. x:ℤ
⊢ x * x ≥ 0
BY

# 2
.....wf.....
ℤ ∈ 𝕌₁
BY
```

For historical reasons, goal labels are sometimes known as *hidden labels*.[3] Labels may be added explicitly using the following tactics.

`AddHiddenLabel` *lab*

Add label *lab* to the current goal.

`AddHiddenLabelAndNumber` *lab i*

Add label *lab* to the current goal along with the integer label *i*.

`RemoveHiddenLabel`

Remove the label from the current goal.

Note that the goal label created by `AddHiddenLabelAndNumber ‘x‘ 1` is ⌞x⌟ and not ⌞x 1⌟, although it will be displayed as the latter. The number is considered as strictly separate from the label and will only be considered by tactics that discriminate on the number as well. Removing a label is equivalent to adding the label `main`. This label will not be displayed, since unlabeled goals are usually considered main goals.

To make subsequent tactics discriminate on labels one usually applies the tacticals described in Section 8.8.2 below. For convenience, labels are divided into the classes *main* and *aux*. The discriminating tacticals allow one to select either subgoals with a particular label, or subgoals of one of these two classes.

8.2.4 Soft Abstractions

In most proofs tactics do not deal with basic expressions of Nuprl's logic but with user-defined concepts. Before applying generic tactics to these, the corresponding abstractions need to be unfolded first. To prevent unwanted unfolding of abstractions, tactics usually do not unfold abstractions unless they are designated as *soft*.

Some tactics treat soft abstractions as being transparent, that is they behave as if all soft abstractions had first been unfolded. In practice, those tactics only unfold soft abstractions when they need to and for the most part are careful not to leave unfolded soft abstractions in the subgoals that they generate.

Specific tactics that unfold soft abstractions are `MemCD`, `EqCD`, `NthHyp`, `NthDecl` (Section 8.3.1), `Eq` (Section 8.3.3), `Inclusion`. (Section 8.7), the forward and backward chaining tactics (Section 8.4), and the atomic rewrite conversions based on lemmata and hypotheses (Section 8.9.2).

Table 8.1 lists the most important soft abstractions in Nuprl's standard libraries. The logic abstractions (`and`, `or`, `implies`, `exists`, `all`) are made soft because the well formedness rule for the underlying primitive term is simpler and more efficient than the well formedness lemma would

[3] In Nuprl 3 labels used not to be visible when editing proofs with the proof editor.

`member`	$t \in T$	$\equiv$	$t = t \in T$	`and`	$A \vee B$	$\equiv$	$A \times B$
`nequal`	$x \neq y \in T$	$\equiv$	$\neg(x = y \in T)$	`or`	$A \wedge B$	$\equiv$	$A + B$
`ge`	$i \geq j$	$\equiv$	$j \leq i$	`implies`	$A \Rightarrow B$	$\equiv$	$A \rightarrow B$
`gt`	$i > j$	$\equiv$	$j < i$	`rev_implies`	$A \Leftarrow B$	$\equiv$	$B \Rightarrow A$
`lelt`	$i \leq j < k$	$\equiv$	$(i \leq j) \wedge (j < k)$	`iff`	$A \Leftrightarrow B$	$\equiv$	$(A \Rightarrow B) \wedge (A \Leftarrow B)$
`lele`	$i \leq j \leq k$	$\equiv$	$(i \leq j) \wedge (j \leq k)$	`exists`	$\exists x{:}A.\ B_x$	$\equiv$	$x{:}A \times B_x$
`prop`	$\mathbb{P}_i$	$\equiv$	$\mathbb{U}_i$	`all`	$\forall x{:}A.\ B_x$	$\equiv$	$x{:}A \Rightarrow B_x$

Table 8.1: Soft abstractions in NUPRL's basic libraries

be. The softness is also useful when one wishes to blur the distinction between propositions and types, for example when reasoning explicitly about the inhabitants of propositions. `member`, `nequal`, `rev_implies`, `ge` and `gt` are soft principally because it can simplify matching.

Abstractions are not soft by default. They can be declared soft or hard by supplying their opids to the functions

`add_soft_abs` *abs*

Declare the abstractions in the token list *abs* as soft.

`remove_soft_abs` *abs*

Declare the abstractions in the token list *abs* as hard.

Instances of these functions are usually kept in ML objects in close proximity to the abstraction definitions that they are declaring soft. For an example use of `add_soft_abs`, see the object `soft_ab_decls` in the `core_2` theory.

8.2.5 Universal Formulas

Many of NUPRL's tactics work on a specific subclass of logical formulas generated by the grammar

$$P \quad ::= \quad \forall x{:}T.\ P \mid P_a \Rightarrow P \mid P \Leftarrow P_a \mid P \wedge P \mid P \Leftrightarrow P \mid C$$

where T is a type, and C is a propositional term not of the above form. We call these *universal formulas*, *positive definite formulas*, or *Horn clauses*. Formulas generates without the $\wedge$ and $\Leftrightarrow$ connectives are also called *simple universal formulas*. We call the proposition C a *consequent* and each P_a an *antecedent*. Occasionally we refer to the types T as *type antecedents*.

We view a universal formula as being composed of several simple formulas, one for each consequent. The simple components are numbered from 1 up, starting with the leftmost consequent.

Such formulas are the standard way of describing *derived rules of inference*, and are used as such by the forward and backward chaining tactics (see Section 8.4). Often, a consequent C of a formula will be an equivalence relation, in which case the formula can be used as a rewrite rule by the rewrite package (see Section 8.6).

Occasionally, one has a universal formula where the outermost constructor of C is also one of the constructors that make up the universal formula. In this case, one can surround C by a *guard* abstraction to designate it as consequent of the formula. A guard abstraction takes a single subterm as argument and unfolds to this subterm. The tactics that take apart universal formulas recognize and automatically remove guard abstractions, so the user rarely has to explicitly unfold them.

8.3 Basic Tactics

8.3.1 Single-Step Decomposition

Single-step decomposition tactics invoke the primitive *formation*, *equality*, and *elimination* rules of Nuprl's logic described in Appendix A.3. These rules describe how types and their members can be formed and when two types or members are equal by decomposing the corresponding terms in the hypotheses or the conclusion of a goal into smaller fragments. Structurally the effect is always the same, as the top-level terms are analyzed and their sub-terms will occur in the subgoals.

Each single-step decomposition tactic covers a large collection of primitive inference rules using a single name. The tactic name indicates which part of a hypothesis or conclusion will be decomposed.

`D` *c*

Decompose the *outermost connective* of clause *c*. `D` can take several optional arguments:

- A ‘`universe`‘ argument, usually supplied using the `At` tactical.
- A ‘`t1`‘ argument for a term (using `With`). This argument is necessary when decomposing a hypothesis with an outermost universal quantifier, or a conclusion with an outermost existential quantifier.
- ‘`v1`‘ and ‘`v2`‘ arguments for new variable names (using `New`. These are useful if one is not satisfied with the system supplied variable names.
- An ‘`n`‘ argument to select a subterm (using `Sel`). This is necessary when applying `D` to a disjunct in the conclusion.

Usually `D` unfolds all top level abstractions and applies the appropriate primitive (formation or elimination) rule. It is somewhat intelligent with instances of *set* and *squash* terms.

`ID` *c*

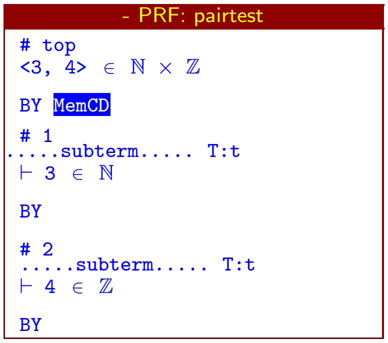
Intuitionistically decompose clause `c`. This behaves as `D` does, except that when decomposing a function, a universal quantifier, or an implication, in a hypothesis, the original hypothesis is left intact rather than thinned.

`MemCD`

Decompose the immediate subterm of a *membership term* in the *conclusion*.

For primitive terms `MemCD` uses the appropriate primitive equality rule. For abstractions, it tries to use an appropriate well-formedness lemma. Soft abstractions will be unfolded if there is no appropriate well-formedness lemma. A subgoal corresponding to the *n*-th subterm will be labeled with label `subterm` and number *n*. Other subgoals are labeled `wf`.
An example application of `MemCD` is shown on the right.

Well-formedness lemmata for a term *t* with operator identifier *opid* should have the name *opid*`_wf` and consist of a simple universal formula with consequent $\lfloor t \in T \rfloor$. Usually, the subterms of *t* should be all variables, but constants are acceptable too. If more than one lemma is needed, the lemma names should be distinguished by suffices to the *opid*`_wf` root.



```
- PRF: pairtest
# top
<3, 4> ∈ ℕ × ℤ

BY MemCD

# 1
.....subterm..... T:t
⊢ 3 ∈ ℕ

BY

# 2
.....subterm..... T:t
⊢ 4 ∈ ℤ

BY
```

Usually `MemCD` attempts to use lemmata in *reverse dependency order* (i.e. the later ones do not refer to the former and are often more general) but different orders may be specified if needed. If the conclusion is $a \in A$ where a is an instance of t and A does not match any of the T of the lemmata, then `MemCD` tries matching a against the term t of the last lemma. If this succeeds and

Tactic	In the hypotheses	In the conclusion
UnivCD		$\forall \ \Rightarrow \qquad \rightarrow$
GenUnivCD		$\forall \ \Rightarrow \ \wedge \ \Leftrightarrow \ \rightarrow \ \times$
RepD	$\wedge$	$\forall \ \Rightarrow \qquad \rightarrow$
GenRepD	$\wedge$	$\forall \ \Rightarrow \ \wedge \ \Leftrightarrow \ \rightarrow \ \times$
ExRepD	$\exists \ \wedge$	$\forall \ \Rightarrow \qquad \rightarrow$
GenExRepD	$\exists \ \wedge \ \vee$	$\forall \ \Rightarrow \ \wedge \ \Leftrightarrow \ \rightarrow \ \times$

Table 8.2: Iterated decomposition tactics and the connectives they decompose

generates some substitution σ, `MemCD` produces a subgoal $\sigma T \subseteq A$ and tries to use the `Inclusion` tactic (Section 8.7) to prove it.

`MemHD` i

Decompose the immediate subterm of a *membership term* in *hypothesis* i.

Since there are no primitive rules for decomposing equalities in hypotheses, `MemHD` only works on hypotheses when the type of the membership term is a `product` or `function` type. For products, the decomposition generates the first and second projection of the term. For functions, it creates a function application. A `‘t1‘` argument is required in this case.

`MemberEqD` c

Decompose the immediate subterm of a *membership term* in clause c. Works like `MemCD` on the conclusion and like `MemHD` on a hypothesis.[4]

`EqD` c

Decomposes terms which are the immediate subterms of an *equality term* in clause c.

`EqD` is like `MemberEqD`, except that it expects and generates equality terms rather than membership terms. It is good for congruence reasoning and is used extensively by the rewrite package (see Section 8.6).
Commonly used variants are the tactics `EqCD` and `EqHD` i, which work identical to `EqD` on the conclusion and a hypothesis, respectively.

`EqTypeD` c

Decompose just the *type subterm* of an *equality* term in clause c. Only works when the type is a `set` type or is an abstraction that eventually unfolds to a set type.
Commonly used variants are `EqTypeCD` and `EqTypeHD` i

`MemTypeD` c

Decompose just the *type subterm* of a *membership* term in clause c. Only works when the type is a `set` type or is an abstraction that eventually unfolds to a set type.
Commonly used variants are `MemTypeCD` and `MemTypeHD` i

The above tactics cover almost all of the primitive inference rules of Nuprl's logic. Appendix A.3 gives a complete description of all the inference rules and the corresponding tactics.

Several tactics perform iterated decomposition of clauses. Table 8.2 lists the most commonly used ones, together with the logical connectives they decompose. These tactics do not decompose guarded terms. If a guard is encountered in the process of decomposing the conclusion, the guard is removed and decomposition of the conclusion stops.

[4] A better name would have been `MemD`

8.3.2 Structural

Structural tactics invoke inferences that depend only on the syntactical structure of the proof goal but are independent of a particular logic.

`Hypothesis`
: Prove goals of form $\ldots A \ldots \vdash A'$ where the conclusion A' is α-equal to A.

`NthHyp` i
: Prove goals of form $\ldots A \ldots \vdash A'$ where A is the i-th hypothesis and A' is α-equal to A.

`Declaration`
: Prove goals of form $\ldots x{:}T \ldots \vdash x \in T'$ or $\ldots x{:}T \ldots \vdash x{=}x \in T'$ where T' is α-equal to T.

`NthDecl` i
: Prove goals of form $\ldots x{:}T \ldots \vdash x \in T'$ or $\ldots x{:}T \ldots \vdash x{=}x \in T'$ where $x{:}T$ is the i-th declaration and T' is α-equal to T.

`Contradiction`
: Prove goals where both P and $\neg P$ occur in the hypotheses list.

`Assert` t
: Assert term t as last hypothesis. Generates a `main` subgoal with t asserted, and an `assertion` subgoal to prove t. Logically this inference is often called a *cut*.
: A more general variant is AssertAt i t, which asserts t before hypothesis i.

`Thin` i
: Delete hypothesis i.

`MoveToHyp` i j
: Move hypothesis i to before hypothesis j.

`MoveToConcl` i
: Move hypothesis i into the conclusion. If the goal has the form $\ldots A \ldots \vdash C$, where A is the i-th hypothesis, then `MoveToConcl` generates the `main` subgoal $\ldots \vdash A \Rightarrow C$. If the i-th hypothesis is a declaration $x{:}T$, it generates the `main` subgoal $\ldots \vdash \forall x{:}T.C$.
: `MoveToConcl` first invokes itself recursively on any hypothesis that might depend on hypothesis i.

`MoveDepHypsToConcl` i
: Use `MoveToConcl` to move all hypotheses that use the variable declared by hyp i into the conclusion. Hypothesis i itself will not be moved.

`RenameVar` v i
: Rename the variable declared in hypothesis i to v.

`RenameBVars (`σ`:(var#var) list)` c
: Rename all occurrences of bound variables in clause c using the substitution σ.

The following two tactics are usually used inside other tactics.

`Id`
: The identity tactic, which does not change the proof goal

`Fail`
: The tactic that always fails.

8.3.3 Decision procedures

Decision procedures use special-purpose reasoning to decide problems within a specific limited application domain. In many cases, they analyze the problem by translating it into a different problem domain (e.g. the problem of finding cycles in a directed graph) for which there are well-known decision algorithms. To ensure consistency with type theory, some of these procedures have to generate proof tactics that validate the result of the analysis and create appropriate subgoals if necessary. For `Arith` and `Eq`, the consistency of the decision procedure has been proven externally and the procedures are implemented as elementary inference rules.

Decision procedures require the well-formedness of the components used during the analysis to be proven as separate subgoals, as this is a purely type-theoretical issue that cannot be addressed by the decision algorithm.

8.3.3.1 Logical reasoning

`ProveProp`

Prove a goal that involves only simple propositional reasoning.
The proof strategy is basically a classical tableau prover for propositional logic, which exhaustively decomposes propositions and seeks for applications of the `Hypothesis` tactic. Because NUPRL sequents only allow one conclusion (rather than many as in tableau calculi) the tactic has to do 'or' branching and backtracking when it tackles an $\vee$ conclusion or an $\Rightarrow$ or $\neg$ hypothesis. It fails if not all main goals can be solved.

The tactic is not complete for intuitionistic propositional logic, because it always thins $\Rightarrow$ and $\neg$ hypotheses that are decomposed. Common variants of `ProveProp` are

- `ProvePropWith` T, which tries running the tactic T before abandoning a search path in an 'or' branch and continues the search on any `main` subgoal that T creates.
- `ProveProp1`, which leaves main subgoals at 'or' branching points of the search for a solution when the search down every branch fails.

`JProver`

Prove a goal that involves only first-order reasoning.
`JProver` [SLKN01] is a complete theorem prover for first-order intuitionistic logic that is based on a strategy called the connection method [KO99]. Upon success it generates a sequent proof for the proof goal [KS00] that may be inspected by the user. Since first-order logic is undecidable, `JProver` will not terminate if the goal cannot be proven and must be interrupted.
`JProver` is run as an *external* prover, which means that the METAPRL proof engine must be connected to NUPRL before invoking `JProver`.

8.3.3.2 Exploiting properties of relations

`Eq`

Prove goals of form $H \vdash s = t \in T$ using hypotheses that are equalities over T and the laws of reflexivity, commutativity and transitivity.
`Eq` also uses hypotheses that are equalities over a type T' when $T = T'$ can be deduced from other hypotheses using reflexivity, commutativity and transitivity.

`RelRST`

Prove goals by exploiting common properties of binary relations, including reflexivity, symmetry, transitivity, irreflexivity, antisymmetry, and linearity.
The heart of `RelRST` is a routine that builds a directed graph based on the binary relations in a sequent and finds shortest paths in the graph. It can also handle strict order relations and relations with differing strengths.

`RelRST` uses the the same database on relations and some of the same lemmata as the rewrite package (see Section 8.6). In addition, it relies on library lemmata of the following forms.

- *Irreflexivity* lemmata, which should be named *opid-lt*`_irreflexivity` and have form
  $\forall x_1{:}T_1 \ldots x_m{:}T_m.\forall y{:}S.A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow \neg(y < y)$
- *Antisymmetry* lemmata, which should be named *opid-le*`_antisymmetry` and have form
  $\forall x_1{:}T_1 \ldots x_m{:}T_m.\forall y,y'{:}S.A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow y \leq y' \Rightarrow y' \leq y \Rightarrow y = y' \in S$
- *Complementing* lemmata, which should be named *opid-le*`_complement` and have form
  $\forall x_1{:}T_1 \ldots x_m{:}T_m.\forall y,y'{:}S.A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow \Rightarrow \neg(y \leq y') \Rightarrow y' < y$
  or be named *opid-lt*`_complement` and have form
  $\forall x_1{:}T_1 \ldots x_m{:}T_m.\forall y,y'{:}S.A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow \Rightarrow \neg(y < y') \Rightarrow y' \leq y$

where *opid-lt* and *opid-le* are the operator identifiers of the relations $<$ and $\leq$.

`RelRST` generalizes the equality decision procedure used in previous versions of Nuprl that could only handle such reasoning with the equality relation. Examples of its use can be found in the theory of integer divisibility within the theory `num_thy_1`.

8.3.3.3 Integer arithmetic

`Arith`

Prove goals of the form $H \vdash C_1 \vee \ldots \vee C_m$ by a restricted form of arithmetic reasoning. Each C_i must be an *arithmetic relation* over the integers built from $<$, $\leq$, $>$, $\geq$, $=$, $\neq$, and negation.

`Arith` knows about the ring axioms for integer multiplication and addition, the total order axioms of $<$, the reflexivity, symmetry and transitivity of equality, and a limited form of substitutivity of equality. As a convenience `Arith` will attempt to prove goals in which not all of the C_i are arithmetic relations; it simply ignores such disjuncts.

The heart of `Arith` is a procedure that translates the sequent into a directed graph whose egdes are labelled with natural numbers and finds positive cycles in that graph [Cha82].

`RepeatEqCDForArith`

Apply arithmetic equality reasoning if the conclusion is $a{=}b \in T$ and a and b arithmetically simplify to same expression.[5] `RepeatEqCDForArith` first decomposes a and b using `EqCD` and then applies `Arith` to subgoals containing integer equalities.

`SupInf`

Solve linear inequalities over integers and subtypes of integers.

The algorithm used in `Arith` cannot solve general sets of linear inequalities over the integers, though such problems are abundant. `SupInf` uses an adaptation of Bledsoe's *Sup-Inf* method [Ble75] for solving integer inequalities. While this method is only complete for the rationals, it is sound for the integers and does work well in practice.

`SupInf` converts the sequent into a conjunction of terms of the form $0 \leq e_i$ where each e_i is a linear expression over the rationals in variables $x_1 \ldots x_n$ and determines whether or not there exists an assignment of values to the x_j that satisfies the conjunction. The algorithm works by determining upper and lower bounds for each of the variables in turn.

`SupInf` identifies counter-examples if it fails. These can be viewed by looking at value of the ML variable `supinf_info`. The value gives a list of bindings of variables in the goal for the counter-

[5] Arithmetic simplification applies only subterms that involve the basic arithmetic operators `+`, `-`, `*`, `/` and `rem`. Currently simplification involving `/` and `rem` does not work and has been disabled.

example. If SupInf finds an integer counterexample, then the goal is definitely unprovable. If a rational counter-example is given, then SupInf is unsure whether the goal is true or not.

SupInf'

Like SupInf, but tries inferring additional arithmetic information about the non-linear terms in arithmetic expressions. The information on a non-linear term t is gathered in two ways:

1. If standard type-inference returns a subtype of $\mathbb{Z}$ for t, then the predicate information from the subtype is added.
2. Information may be gathered from *arithmetic property lemmata*, i.e. lemmata of the form

   $$\forall x_1{:}T_1 \ldots x_m{:}T_m.A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow C$$

   where C is constructed from $\wedge$, $\vee$, and standard arithmetic relations over integer subtypes built from $<$, $\leq$, $>$, $\geq$, $=$, $\neq$, and their negations. To apply the lemma, *match handles* are selected from C, i.e. terms occuring in arithmetic relations in C that contain all the free variables contained in C and the A_i. If, after matching a match handle with t, all the instantiated A_i are equal to hypotheses of the sequent, then the instantiated clause C will be added as information.[6]

   Arithmetic property lemmata are identified by invoking the ML function

   add_arith_lemma *lemma-name*.

SupInf' is still under development and should be used with caution when combined with type-checking tactics: it may get invoked unendingly on subgoals derived from ones that it created.

8.3.4 Autotactics

Autotactics are used primarily for typechecking and well-formedness goals. They should not be used for other non-trivial proof goals, as their behavior on such goals is somewhat unpredictable.

Trivial

Completely prove a goal by applying various steps of trivial reasoning.
Trivial reasoning includes Hypothesis, Declaration, Contradiction, and Eq. Trivial also proves goals of the form $H \vdash$ True, $H_1 \ldots$ False $\ldots H_n \vdash C$, and $H_1 \ldots$ Void $\ldots H_n \vdash C$.

Auto

Repeatedly apply the following tactics until no further progress is made.

- Trivial
- GenExRepD
- MemCD for (non-recursive) member conclusions and EqCD on reflexive equality conclusions.
- Arith
- RepeatEqCDForArith
- EqTypeCD if the conclusion is $a \in T$ or $a{=}b \in T$ and T is a subset of $\mathbb{Z}$.

Auto and its variants frequently encounter the same goals over and over again, so solved proof goals will be cached. Common variants of Auto are

- StrongAuto, which also tries MemCD and EqCD on *recursive* primitive terms;
- SIAuto, which also tries using the SupInf tactic; and
- Auto', which uses SupInf' instead of Arith.

[6] The condition that A_i must be equal to some hypothesis is too strict and may be relaxed in the future.

8.4 Forward and Backward Chaining

Forward and backward chaining means treating a component of a universal formula (see Section 8.2.5) as derived inference rule. *Backward chaining* involves matching the conclusion of the goal against the consequent of a universal formula, which leaves the instantiated antecedents of the universal formula as new subgoals. *Forward chaining* involves matching hypotheses of the goal against antecedents of a universal formula and asserting the instantiated consequent of the universal formula as a new hypothesis. A simplified version of forward chaining is *instantiating* the universal formula by explicitly providing a list of terms to be substituted for the quantified variables.

Chaining tactics consult universal formulas can either be found in the hypotheses of the current proof goal or as lemma in the library. Therefore each tactic comes in two versions.

`InstHyp [`t_1`;..;`t_n`]` *i*
`InstLemma` *name* `[`t_1`;..;`t_n`]`

Instantiate hypothesis *i* (or lemma *name*) with terms $t_1 \ldots t_n$. If the lemma has m distinct level expressions, the first m terms should be level expressions to substitute for these.[7]

`InstConcl [`t_1`;..;`t_n`]`

Instantiate existential quantifiers in the conclusion with terms $t_1 \ldots t_n$.

`FHyp` *i* `[`h_1`;..;`h_n`]`
`FLemma` *name* `[`h_1`;..;`h_n`]`

Forward chain through hypothesis *i* (or lemma *name*) matching its antecedents against any of the hypotheses $h_1 \ldots h_n$. The order of the h_j is immaterial: the tactics try all possible pairings of hypotheses with antecedents. If there are more antecedents that hypotheses listed, the antecedents not matched will manifest themselves as new subgoals to be proved.
The main subgoal with the consequent asserted is labelled `main`. Unmatched antecedents are labelled `antecedent` and the rest are labelled `wf`. Aliases are `FwdThruLemma` and `FwdThruHyp`.

Forward chaining can take an optional argument (c.f. Section 8.2.2), supplied by using the `Sel` tactical, to select a specific component of a conjunction or equivalence in a universal formula. An argument of `-1` forces the tactic to treat the whole subformula as simple.

`BHyp` *i*
`BLemma` *name*

Backward chain through hypothesis *i* (or lemma *name*) matching its consequent against the conclusion of the goal. Subgoals corresponding to antecents of the lemma (hyp) are labelled with `antecedent`. The rest are labelled `wf`. Aliases are `BackThruLemma` and `BackThruHyp`.

An explicit list of variable bindings can be supplied to backward chaining as optional argument by using the `Using` tactical. This argument is necessary if some of the variable bindings cannot be inferred by matching. For instance

```
Using ['n'.'3'] (BackThruLemma 'int_upper_induction')
```

would bind the variable `n` in the lemma `int_upper_induction` to the value `3`.

`BHypWithUnfolds` *i* *as*
`BLemmaWithUnfolds` *name* *as*

Backward chain through hypothesis *i* (or lemma *name*) while unfolding the abstractions in *as*.

[7] To inject a level expression L into a term list one has to invoke the special term `parameter` in the term editor.

`Backchain` *bc_names*
`CompleteBackchain` *bc_names*

Repeatedly try backchaining using lemmata named in *bc_names* in the order given. `Backchain` leaves alone any subgoals which do not match the consequent of any of the lemmata while `CompleteBackchain` backtracks in the event of any such subgoal coming up.

In addition to lemma names, *bc_names* may contain a few special names:

- An positive integer i, indicating that hypothesis i shall be consulted.
- `‘hyps‘`, indicating that all hypothesis should be consulted in the order in which they occur. Hypotheses that declare variables will be ignored.
- `‘rev_hyps‘`, indicating that all hypothesis should be consulted in reverse order.
- `‘new_hyps‘`, indicating that all new hypotheses introduced by backchaining should be consulted, beginning with the least recent.
- `‘rev new_hyps‘`, indicating that all new hypotheses introduced by backchaining should be consulted, beginning with the most recent.

Common variants of backchaining are `HypBackchain` and `CompleteHypBackchain` which perform backchaining with the arguments `[‘rev_new_hyps‘; ‘rev_hyps‘]`.

8.5 Case Splits and Induction

Case split and induction tactics analyze conclusions that depend on finite or enumerable alternatives. The general way to do this is to backchain through an appropriate lemma (see e.g. the lemmata `int_upper_ind` and `int_seg_ind` at the end of the `int_2` theory). To use these lemmata, one must ensure that the outermost universal quantifier in the conclusion corresponds to the type of the induction.

The following tactics are good for a few common cases. They expect the variable the induction / case-split is being done over to be declared in some hypothesis.

`BoolCases` i

Do a case split over a boolean variable declared in hypothesis i. Generates two subgoals, labelled `truecase` and `falsecase`, where the boolean variable is replaced by `tt` or `ff`.

`Cases` `[`t_1`;..;`t_n`]`

Perform an n-way case split over the terms t_i as follows:

$$\begin{array}{l}
H \vdash C \\
\quad \textbf{by}\ \texttt{Cases}\ [t_1;..;t_n] \\
\qquad \texttt{...assertion...}\ H \vdash t_1 \vee .. \vee t_n \\
\qquad\qquad\qquad\qquad\quad H,\ t_1 \vdash C \\
\qquad\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\qquad\quad H,\ t_n \vdash C
\end{array}$$

`Decide` P

Perform a case split over a *decidable* proposition P and its negation. Like `Cases [`P`;`$\neg P$`]`, but immediately runs the tactic `ProveDecidable` tactic on the first subgoal $\vdash\ P \vee \neg P$, which may generate wellformedness subgoals with labels in the `aux` class. If `ProveDecidable` fails then `Decide` fails too.

Because of the constructive nature of Nuprl’s type theory $P \vee \neg P$ is not true for every proposition P.

`IntInd` *i*

Perform integer induction on hypothesis *i*, generating three subgoals labelled `upcase`, `basecase` and `downcase`. `IntInd` is a little smarter than the primitive rule `intElimination` (see Appendix A.3.8) in that it first moves any hypotheses that depend on the induction variable to the conclusion and maintains the name of the induction variable.

`NatInd` *i*

Perform natural-number induction on hypothesis *i*, which must contain a declaration of type $\mathbb{N}$, generating two subgoals labelled `upcase` and `basecase`. `NatInd` first moves any depending hypotheses to the conclusion and maintains the name of the induction variable.

`NSubsetInd` *i*

Perform induction on a subrange of the natural numbers. Hypothesis *i* must contain a declaration of type $\mathbb{N}$, $\mathbb{N}^+$, $\{i\ldots\}$, or $\{i\ldots j^-\}$. `NSubsetInd` generates two main subgoals labelled `upcase` and `basecase` an many `aux` subgoals which should always be easily solvable by `Auto`.

`CompNatInd` *i*

Perform *complete* induction on hypothesis *i*, which must contain a declaration of type $\mathbb{N}$.

`ListInd` *i*

Perform list induction on hypothesis *i*, which must contain a declaration of type T `list`, generating two subgoals labelled `upcase` and `basecase`. `ListInd` is a little smarter than the primitive rule `listElimination` (see Appendix A.3.10) in that it first moves any depending hypotheses to the conclusion and maintains the name of the induction variable.

The theory `well_fnd` has some definitions for *well-founded* induction. In particular it defines the tactic `Rankind`. This is useful when you know how to do induction over some type A and you want to perform induction over a type B using some *rank* function which maps elements of B to elements of A. The tactic is described in the objects `inv_image_ind_tac` and `rank_ind`.

The theory `bool_1` defines various tactics for case splitting on the value of boolean expressions in the conclusion such as `BoolCasesOnCExp` and `SplitOnConclITE`. View the theory for details.

8.6 Simple Rewriting

Rewriting is the process of transforming goals and hypotheses into equivalent ones. In Nuprl, rewrite tactics are based on unfolding and folding abstractions, applying primitive reductions, and using equivalences in lemmata and hypotheses of the form $\forall x_1{:}T_1\ldots x_i{:}T_i.\ a = b$.

Nuprl's rewrite package (see Section 8.9) provides a collection of ML functions for creating rewrite rules and applying them in various fashions. The rewriting tactics in this section are sufficient in many situations while hiding the complex conversion language of the rewrite package.

8.6.1 Folding and Unfolding Abstractions

`Unfolds` *as c*

`Unfold` *a c* ≡ `Unfolds [`*a*`]` *c*

Unfold all visible occurrences of abstractions listed in the token list *as* in clause *c*. `Unfolds` fails if clause *c* contains none of the abstractions listed in *as*.

`RepUnfolds` *as c*

Repeatedly try unfolding any occurrences of abstractions listed in the token list *as* in clause `c`.

`RecUnfold` *a c*

Unfold all visible occurrences of the recursive definition of *a* in clause *c*.

`Folds` *as* *c*
`Fold` *a* *c* ≡ `Folds [`*a*`]` *c*

Fold all visible occurrences of abstractions listed in the token list *as* in clause `c`. `Folds` fails none of the subterms of clause *c* matches the right hand side of an abstraction listed in *as*.

`RecFold` *a* *c*

Try to fold an instance of the recursively defined term *a* in clause *c*.

8.6.2 Evaluating Subexpressions

`Reduce` *c* ≡ `AbReduce` *c*

Repeatedly contract all primitive and abstract redices in clause *c*.

The `Reduce` tactic can take an optional *force* argument:

`With` *force* `(Reduce` *c*`)`

only reduces those redices with strength less than or equal to *force*. Details about defining abstract redices and setting the strength of redices can be found in Section 34.

To reduce a specific subterm of a clause, one may apply the tactic `ReduceAtAddr` *address* *c*, where *address* is a list of integers describing the exact address of the subterm in the term tree of clause *c*. Applying this rule is only recommended for advanced users who are very familiar with the term structure of Nuprl expressions.

`ArithSimp` *c*

Arithmetically simplify clause *c*.
This creates a `main` subgoal with clause *c* rewritten in arithmetical canonical form and an `aux` subgoal stating the equivalence between the original clause and the rewritten one.

8.6.3 Substitution

Nuprl's logic contains a few rules for carrying out simple kinds of substitutions. These rules often generate fewer and easier-to-solve well-formedness goals than the rewrite package and are accessible through the tactics described here.

`Subst` *eq* *c*

If the term *eq* has the form $t_1 = t_2 \in T$ then replace all occurrences of t_1 in clause *c* by t_2.
Three subgoals are generated: an `equality` subgoal to prove that $t_1 = t_2 \in T$, a `main` subgoal with the substitution carried out, and a `wf` subgoal to prove functionality of the clause (see the rule `substitution` in Section A.3.5).

`HypSubst` *i* *c*
`RevHypSubst` *i* *c*

Run the `Subst` tactic using the equality proposition in hypothesis *i* (in reversed order). This generates only a `main` and a `wf` subgoal.

`SubstClause` *t* *c*

Replace clause *c* with term *t*. This generates a `main` subgoal and an `equality` subgoal (see the rule `hyp_replacement` in Section A.3.16).

Token	Rule
i	Use hypothesis *i* as an left-to-right rule
i`<`	Use hypothesis *i* as an right-to-left rule
name	Use lemma *name* as an left-to-right rule
name`<`	Use lemma *name* as an right-to-left rule
`r:`*id*	Reduce redex with operator identifier *id*
`r*`	Reduce any redex
`r*`*force*	Reduce any redex with force *force*
`u:`*id*	Unfold abstraction with operator identifier *id*
`f:`*id*	Fold abstraction with operator identifier *id*

Table 8.3: Format of Tokens in Rewrite Control Strings

8.6.4 Generic Rewrite Tactics

The tactics `RWW` and `RWO` subsume the above tactics by providing uniform access to all kinds of rewrite rules. They take a control string to specify the rewrite rules to use. The control string should be a whitespace-separated list of tokens as specified in Table 8.3.

`RWW "`*ctl-str*`"` *c*

Repeatedly apply rewrite rules specified by *ctl-str* to all subterms of clause *c* until no further progress is made.

`RWO "`*ctl-str*`"` *c*

Apply rewrite rules specified by *ctl-str* in one top-down pass over clause *c*. `RWO` does not go into subterms of terms that result from rewriting a subterm of *c*.

In some cases, `RWW` and `RWO` generate more subgoals than the more specific tactics, as they are implemented in a different fashion.

8.7 Miscellaneous Tactics

Type Inclusion

`Inclusion` *i*

Prove goals of the form $\ldots,\ i.\ x{:}T, \ldots \vdash x \in T'$ or $\ldots,\ i.\ t \in T, \ldots \vdash t \in T'$, where either types T and T' are equivalent or T is a proper subtype of T'. `Inclusion` also solves similar goals where one or both of the membership terms are replaced by equality terms.

The specific kinds of relations between T and T' that `Inclusion` currently handles are roughly:

- T and T' are the same once all soft abstractions are unfolded.
- T and T' are both universe or prop terms and the level of T is no greater than the level of T' for any instantiation of level variables.
- T and T' are each formed by using subset types, and both have some common superset type. In this case `Inclusion` tries to show that the subset predicates of T' are implied by the subset predicates of T together with other hypotheses.
- T and T' have the same outermost type constructor. In this case, the inclusion goal is reduced to one or more inclusion goals involving the immediate subterms of T and T'. Currently works for function, product, union and list types.
- T is a subtype of T' according to a lemma in the library.

For the inclusion reasoning involving subset types to work, one has to supply information about abstractions involving subset types using the function `add_set_inclusion_info`. The theory `int_1` contains several examples of the use of this function.

Squash Stability and Hidden Hypotheses

The constructivity of Nuprl's logic manifests itself in the fact that the decomposition of sets in hypotheses (rule `setElimination` in Section A.3.12) results in a subgoal containing the predicate part of the set term as a *hidden* hypothesis (the rule `quotient_equalityElimination` in Section A.3.14 has a similar effect). A hidden hypothesis does not contribute to the computational content of a proof (i.e. the term inhabiting its conclusion) and is therefore not immediately usable. However, there are ways in which it might become usable later.

A hidden hypothesis P in a proof goal can be unhidden if either P or the conclusion of the goal is *squash stable*, which roughly means that it is possible to determine the computational content if one knows that there is one (in the classical sense). Squash stability is defined in the theory `core_2` as $\downarrow P \Rightarrow P$, where the proposition $\downarrow P \equiv \{x{:}\mathsf{Unit}|P\}$ (read 'squash P') is true exactly when P is true, but has no computational content.

Squash stability can be inferred for many predicates using the tactic `ProveSqStable`, which is not called by the `D` tactic because it can be rather slow. Instead, hypothesis can be unhidden by applying one of the following tactics.

`Unhide`

Try to unhide hidden hypotheses, first by checking whether the conclusion is squash stable and then, if this fails, by checking each hidden hypothesis separately for squash stability.

`Unhide` applies the tactics `UnhideAllHypsSinceSqStableConcl`, which tries to prove the conclusion squash stable using `ProveSqStable` and unhides all hidden hypotheses if this succeeds, and UnhideSqStableHyp i, which tries to prove the hidden hypothesis i squash stable.

`AddProperties` i

Add the predicate part of the set type underlying an abstraction A in hypothesis i as a new hypothesis immediately after i.

Hypothesis i should be declaration of form A or a proposition of form $t \in A$ or $t = t' \in A$, where A is an abstraction with an associated *property lemma* of the form $\vdash \forall x{:}T.\ \forall y{:}A.P[x,y]$.

`GenConcl` $\llcorner t\texttt{=}v \in T \lrcorner$

Generalize occurrences of t as subterms of the conclusion to the variable v. This adds new hypotheses declaring v to be of type T and stating $t\texttt{=}v \in T$.

`Fiat`

If you about to give up hope on a theorem, this tactic is guaranteed to provide satisfaction.
`Fiat` uses Nuprl's `because` rule, which should be used for experimental purposes only. Nuprl's library has a mechanism to detect uses of this rule while checking a theory for consistency.

8.8 Tacticals

Tacticals are functions for converting tactics into new one. Apart from injecting optional arguments as described in Section 8.2.2, they are most commonly used for composing tactics. 2-ary tacticals are often written in infix form and distinguished from others by having the first part of their name in all capitals. Infix tacticals always associate to the left.

8.8.1 Basic Tacticals

T_1 `THEN` T_2

Apply T_1 and then run T_2 on all the subgoals generated by T_1.

T `THENL` `[`T_1`;...;`T_n`]`

Apply T and then run T_i on the i-th subgoal generated by T. T must create exactly n subgoals.

T_1 `ORELSE` T_2

Apply T_1. If it fails, run T_2 instead.

`Try` T $\equiv$ T `ORELSE Id`

Apply T. If it fails, leave the proof unchanged.

`Complete` T

Apply T but fail if T generates subgoals (i.e. does not complete the proof).

`Progress` T

Apply T but fail if T does not change the goal (i.e. makes no progress).

`Repeat` T

Repeat running T on subgoals created by previous applications until no further progress is made.

`RepeatFor` n T

Repeat the application of T exactly n times.

`If` e T_1 T_2

Apply T_1 if e `pf` evaluates to `true`, where `pf` is the current proof goal. Otherwise, run T_2.

8.8.2 Label Sensitive Tacticals

Label sensitive tacticals tacticals allow one to apply a tactic only to goals with a particular label, or to goals of one of the classes *main*, *aux*, and *predicate* (c.f. Section 8.2.3). For the former, the label associated with the tactic has to match exactly the label of the goal. For the latter, one may use the class names `main`, `aux`, and `predicate` as wild cards.[8]

`IfLab` lab T_1 T_2

If lab matches the label of `pf`, run T_1. Otherwise, run T_2.

`IfLabL [`l_1`,`T_1`;` l_2`,`T_2`; ...;` l_n`,`T_n`]`

Run the first tactic T_i for which l_i matches the label of `pf`. If none of the labels match, leave the proof unchanged.

T_1 `THENM` T_2 $\equiv$ T_1 `THEN IfLab ‘main‘` T_2 `Id`
T_1 `THENA` T_2 $\equiv$ T_1 `THEN IfLab ‘aux‘` T_2 `Id`
T_1 `THENW` T_2 $\equiv$ T_1 `THEN IfLab ‘wf‘` T_2 `Id`

Apply T_1 and then run T_2 on all `main`/`aux`/`wf` subgoals.

T `THENLL [`l_1`,`Ts_1`;` l_2`,`Ts_2`;...;` l_n`,`Ts_n`]`

Apply T and then do the following on each subgoal. Select the first list of tactics Ts_i for which l_i matches the label of the subgoal. If the subgoal also has a number label j, run the jth tactic from Ts_i on it. If it has no number label, run the first tactic listed in Ts_i.

`THENLL` fails if there are not sufficiently many tactics in Ts_i. It runs the `Id` tactic if a subgoal label does not match any of the l_i.

`SeqOnM [`T_1`;...;`T_n`]`

Run the tactics T_1 to T_n on successive `main` subgoals.

[8] Unfortunately `main` is currently used as both a class name and a particular label name, which means there is no way to select only subgoals in with the particular label `main`.

RepeatM T
RepeatMFor n T

Repeat the tactic T on main subgoals (exactly n times).

8.8.3 Multiple Clause Tacticals

Multiple clause tacticals allow to apply a tactic T : int -> tactic to several clauses of a goal with n hypotheses.

On [c_1;...;c_n] T ≡ T c_1 THENM ... THENM T c_n

Run T on the clauses c_1... c_n. If T succeeds on some clause, then On only continues on subgoals created by T that are labelled main.

AllHyps T ≡ On [n;n-1;...;1] (ı. Try (T i))

Try running T on all hypotheses starting with the end of the hypothesis list and working backwards. If T succeeds on some hypothesis, then AllHyps only continues on subgoals created by T that are labelled main.

All T ≡ On [n;n-1;...;1;0] (ı. Try (T i))

Try running T on all hypotheses and then on the conclusion.

OnSomeHyp T ≡ T n ORELSE ... ORELSE T 1

Try running T on one of the hypotheses of the goal, starting with the end of the hypothesis list and working backwards.

The library contains a few abstractions that provide a more elegant notation for the most common combinations of tactics and tacticals.

auto	T ...	≡	T THEN Auto
aux_auto	T ...a	≡	T THENA Auto
siauto	T ...	≡	T THEN SIAuto
autop	T ...'	≡	T THEN Auto'
aux_autop	T ...a'	≡	T THENA Auto'
aux_siauto	T ...as	≡	T THENA SIAuto

8.9 The Rewrite Package

NUPRL's rewrite package is a collection of ML functions for creating rules for rewriting terms into equivalent ones and applying them in various fashions to clauses of a sequent. The package supports rewrite rules involving various equivalence relations – such as the 3-place equality-in-a-type relation, logical bi-implication, the permutation relation on lists, abstractions, and computational equivalence – and takes care of automating proofs that these equivalence relations are respected by rewriting. It also supports rewriting rules involving arbitrary transitive relations such as logical implication and takes care of checking that relevant terms are appropriately monotonic.

The package is based around ML objects called *conversions*, similar to those found in other tactic-based theorem provers such as LCF, HOL, and Isabelle, provide a language for systematically building up rewrite rules in a fashion similar to the way tactics are assembled using tacticals.

8.9.1 Introduction to Conversions

A conversion is a function that transforms a term t into a new term t' that is equivalent to t with respect to some *relation* r. The conversion also produces a *justification* j that describes how to prove that $t\ r\ t'$ holds. The transformation takes place in an *environment* e, which specifies amongst other things the types of the variables that might be free in t. Conversions fail if they are not appropriate for the term they are applied to.

In Nuprl, the type `convn` of conversions is an `ML` concrete type abbreviation for the type

```
env -> term -> (term # reln # just),
```

where `env`, `reln` and `just` are abstract types for *environments* (Section 8.9.1.1), *relations* (Section 8.9.1.2), and *justifications* (Section 8.9.1.3). The language of conversions provides a small set of *atomic conversions* that may be assembled into more advanced conversions using higher order functions called *conversionals*.

Atomic conversions (Section 8.9.2) are either based on direct computation rules (which includes folding and unfolding abstractions) or can be created from lemmata and hypotheses that contain universally quantified formulas with consequent of the form $a\ r\ b$, where the free variables of b are a subset of those in a. Applying a conversion to a term t means either executing the corresponding computation or matching the term t against a and replacing it by an appropriate instance of b. That is, if σ is a substitution of the free variables in a such that $\sigma a = t$, then t will be replaced by $t' = \sigma b$. For instance, rewriting the term $(2\times 3) + 0$ with the relation $x + 0 = x$ means matching $(2\times 3) + 0$ against $x + 0$, which yields a substitution σ that binds x to 2×3. The result of rewriting is the term $\sigma x = 2\times 3$.

Atomic conversions cannot by themselves rewrite subterms of a given term. For instance, applying an atomic conversion to rewrite $(1 + 0)\times 3$ with the relation $x + 0 = x$ fails. *Conversionals* (Section 8.9.4) provide the means for applying conversions to subterms of a term and help controlling the sequence in which atomic conversions are applied to these subterms.

An example of a conversional is `SweepUpC`, which attempts to apply a conversion c to each subterm of a term t, working from the leaves of term t up to its root. Another example is `ORELSEC`, which first tries to apply a conversion c_1 to a term and, if that fails, applies a conversion c_2.

Conversionals rely on a variety of lemmata, which we will describe in Section 8.9.1.4. These lemmata have to state reflexivity, transitivity and symmetry properties of the relation r and congruence properties of the terms making up the clauses that are being rewritten.

A tactic `Rewrite : convn -> int -> tactic` is used for making a conversion applicable to some clause of a proof goal. It takes care of executing the justifications generated by conversions. Section 8.9.1.5 lists common variations on this tactic.

8.9.1.1 Environments

An *environment* is a list of propositions and declarations of variable types that are being assumed. The environment of the conclusion of a sequent is the list of all the hypotheses. The environment of a hypothesis is the list of hypotheses to the left of it. We can also talk about *local environments* of subterms of sequent clauses. For example, in the sequent $x_1{:}H_1, \ldots, x_n{:}H_n \vdash \forall y{:}T.\ B \Rightarrow C$ the local environment for the subterm C in the conclusion is $x_1{:}H_1, \ldots, x_n{:}H_n,\ y{:}T,\ B$. The rewrite conversionals keep track of the local environment each conversion is being applied in, and every conversion takes as its first argument an expression e of type `env` which supplies this local information. The environment information is used by lemma and hypothesis conversions in three ways.

- Declarations of variables in the environment are used to infer types that help to complete matches.
- Propositions in the environment state the assumptions that are necessary for conditional rewrites to go through. For example, if the subterm C in $x_1{:}H_1, \ldots, x_n{:}H_n \vdash \forall y{:}T.\ B \Rightarrow C$ is rewritten by a rewrite rule based on the lemma $\forall z{:}T.\ A[z] \Rightarrow t[z] = t'[z]$ and matching the term C against t results in binding the variable z to a term s, then the subgoal that has to be proven for the rewrite rule to be valid is $x_1{:}H_1,\ \ldots, x_n{:}H_n,\ y{:}T, B, \vdash A[s]$.
- The hypothesis conversions access the hypothesis list via environment terms.

Currently, NUPRL only knows how to extend the environment when descending to the subterms of a ∀-, ∃-, and ⇒-term. For other terms, it does not modify the environment, unless explicitly told so by the user, who may extend the list of environment update functions by applying the function `add_env_update_fun`. Further details can be found in the system file `env.ml`.

8.9.1.2 Relations

The rewrite package supports rewriting with respect to both primitive and user-defined equivalence relations. Some examples are:

- $t \sim t'$, the computational equality relation,
- $t = t' \in T$, the primitive equality relation of NUPRL's type theory,
- $P \Leftrightarrow Q$, the logical bi-implication,
- $i = j \ \mathsf{mod}\ n$, the equality on the integers modulo a positive natural,
- $t =_q t'$, equality of rationals (represented as pairs of integers),
- $l \equiv l'$, the permutation relation on lists.

The package also supports 'rewriting' with respect to any relation that is transitive but not necessarily symmetric or reflexive, such as logical implication[9] ⇒ and order relations, because proofs involving transitive relations and monotonicity properties of terms can be made very similar in structure to those involving equivalence relations and congruence properties.

For each user-defined relation, the user has to provide the rewrite package with lemmata about transitivity, symmetry, reflexivity and *strength*, where a binary relation r over a type T is stronger than a relation r' $a\ r\ b$ implies $a\ r'\ b$ for all $a, b \in T$. These lemmata are used by the package for the justification of rewrites (see Section 8.9.1.4). The user may also provide a declaration that identifies *relation families* and extra properties of relations.

Rewrite relations should be defined as first-order terms with two principal arguments supplied as subterms. Additional parameters should always be positioned before the principal argument subterms. For example, the equality relation $t = t' \in T$ has the type T as additional parameter and the internal structure `equal(T;t;t')`. Relations are most commonly represented as logical propositions (i.e. are of type ℙ). Boolean-valued relations are also accepted, but they have to be wrapped in the `assert` abstraction if used in a context where a logical proposition is expected.

Equivalence relations should be declared by an invocation of

`declare_equiv_rel` *rnam stronger-rnam*

This declares the term with operator identifier *rnam* to be an equivalence relation, and the term with operator identifier *stronger-rnam* to be an immediately stronger equivalence relation. Commonly, there will be only one such declaration for each *rnam* and *stronger-rnam* will be `‘equal‘`. However, multiple declarations for a single equivalence relation are sometimes needed.

[9] Note that treating implication as a rewrite relation leads to a generalization of forward and backward chaining.

Order relations are grouped into *relation families*, i.e. lattices of order and equivalence relations of the form:

$$\begin{tikzcd}
 & \leq \arrow[dr, dash] & & \geq \arrow[dr, dash] & \\
< \arrow[ur, dash] & & = \arrow[ur, dash] & & >
\end{tikzcd}$$

where weaker relations are higher in the lattice, and relations within a family satisfy

$$\begin{array}{lclclcl} a{<}b & \Leftrightarrow & b{>}a & \quad & a{=}b & \Leftrightarrow & a{\leq}b \wedge b{\leq}a \\ a{\leq}b & \Leftrightarrow & b{\geq}a & \quad & a{<}b & \Leftrightarrow & a{\leq}b \wedge \neg(b{\leq}a) \end{array}$$

The converse of an order relation r should always be defined directly in terms of r. For example, the definition of the abstraction `rev_implies` is $P \Leftarrow Q \;\equiv\; Q \Rightarrow P$. The rewrite package assumes that order relations can be inverted by folding and unfolding such definitions. A relation family should be declared using an invocation of

`declare_rel_family` *lt le eq ge gt*

where dummy terms (i.e. the abstraction `dummy()`, which displays as `?`) should be used as place-holders when a member of a family is missing. To simplify such invocations, a user may enter the name `relfam` into a term slot, which will display the template:

```
Relation Family                             Relation Family
< : [lt]                                    < : i < j
≤ : [le]                                    ≤ : i ≤ j
≡ : [eq]                                    ≡ : i = j
≥ : [ge]                                    ≥ : i ≥ j
> : [gt]                                    > : i > j
```

As an example, the invocation to the right shows the declaration of the standard order relations on the integers as relation family. Frequently several order relation families share the same equivalence relation, but there may also be equivalence relations that are not be associated with any order relation family.

The partial order of strengths of relations is the reflexive transitive closure of the strength relation for each family and the equivalence relation declarations. Additional relations between order relations can be declared using

`declare_order_rel_pair` *stronger-rtm weaker-rtm*

For the sake of clarity, all relation declarations should be inserted in ML objects that are positioned after the referred-to relations but before any lemmata that might be accessed by the rewrite package.

8.9.1.3 Justifications

The justification produced by a rewrite rule describes how to prove that the origin and the result of rewriting stand in the relation r. This information is used by the rewrite tactic to generate the corresponding NUPRL proof. There are two types of justifications.

Computational Justifications are lists of precise applications of the forward and reverse direct computation rules. As these are comparatively very fast and generate no well-formedness subgoals, the rewrite package uses these whenever possible.

Tactic Justifications are more generally applicable, but make extensive use of lemmata (see Section 8.9.1.4 below) and often generate many well-formedness subgoals.

Conversions generating both types of justification can be freely intermixed; the system takes care of converting computational justifications to tactic justifications when necessary.

8.9.1.4 Lemma Support

The rewrite package must have access to several kinds of lemmata in order to construct justifications for rewrites. This section describes those lemmata.

Functionality Lemmata give congruence and monotonicity properties of terms. They are required by conversionals like `SubC` to construct justifications for rewriting terms based on the justifications for rewriting the immediate subterms of those terms. A functionality lemma for a term with operator *op* should have the form

$$\begin{aligned}&\forall z_1{:}S_1..z_k{:}S_k.\forall x_1,y_1{:}T_1..x_n,y_n{:}T_n.\ A_1\Rightarrow \ldots \Rightarrow A_m\\ &\quad\Rightarrow x_1 r_1 y_1 \Rightarrow \ldots \Rightarrow x_n r_n y_n \Rightarrow op(x_1;..;x_n)\ r\ op(y_1;..;y_n)\end{aligned}$$

where $k, m{\geq}0$. The universal quantifiers and A's can be intermixed, but the antecedents containing the r_i must come afterward and be in the same order as the subterms of *op*.

If *op* binds variables in its subterms, then these variables should be bound by universal quantifiers wrapped around the appropriate r_i antecedents. For example, the lemma for functionality of $\exists$ with respect to the $\Leftrightarrow$ relation is:

$$\begin{aligned}&\forall A_1, A_2{:}\mathbb{U}_i.\forall P_1{:}A_1\Rightarrow \mathbb{P}_i.\forall P_2{:}A_2\Rightarrow \mathbb{P}_i.A_1 = A_2 \in \mathbb{U}_i\\ &\quad\Rightarrow (\forall x{:}A_1.\ P_1[x] \Leftrightarrow P_2[x])\ \Rightarrow\ \exists x{:}A_1.P_1[x]\ \Leftrightarrow\ \exists x{:}A_2.P_2[x]\end{aligned}$$

To allow conversionals to find functionality lemmata in the library, they should be named `opid_functionality[_index]` where *opid* is the operator identifier of *op* and `_index` is an optional suffix. When more than one functionality lemma is created for a given operator, they must be ordered with the most specific $r_1 \ldots r_n$ first, as conversionals search for functionality lemmata in the order in which they appear.

Functionality lemmata are not needed when all the r_i and r are primitive equalities. In this case functionality information can be derived from the well-formedness lemma for *op*.

Transitivity Lemmata give transitivity information for rewrite relations. They are used to construct the justification in sequencing conversionals like `ANDTHENC` and should have the form:

$$\forall z_1{:}S_1..z_k{:}S_k.\forall x_1, x_2, x_3{:}T.\ A_1\Rightarrow \ldots \Rightarrow A_m \Rightarrow x_1\ r_a\ x_2 \Rightarrow x_2\ r_b\ x_3 \Rightarrow x_1\ r_c\ x_3$$

where $k, m{\geq}0$ and r_c should be the weaker of r_a and r_b. Transitivity lemmata should be named `opid-of-`r_c`_transitivity[_index]`. Transitivity lemmata are not needed for primitive equality relations.

Weakening Lemmata extend the usefulness of the transitivity and functionality lemmata. They should have the form

$$\forall z_1{:}S_1..z_k{:}S_k.\forall x_1, x_2{:}T.\ A_1\Rightarrow \ldots \Rightarrow A_m \Rightarrow x_1\ r_a\ x_2 \Rightarrow x_1\ r_b\ x_2,$$

where $k, m{\geq}0$ and r_b is weaker than r_a, and be named `opid-of-`r_b`_weakening[_index]`. Weakening lemmata are required for all reflexive relations r_b with r_a being equality.

Inversion Lemmata are used by the `Rev*` atomic conversions and in conjunction with weakening, transitivity, and functionality lemmata when these mix order and equivalence relations. They are required for equivalence relations, but not for equality or order relations. Inversion lemmata should have the form

$$\forall z_1{:}S_1..z_k{:}S_k.\forall x_1, x_2{:}T.\ A_1\Rightarrow \ldots \Rightarrow A_m \Rightarrow x_1\ r\ x_2 \Rightarrow x_2\ r\ x_1$$

where $k, m{\geq}0$ and should be named `opid-of-`r`_inversion`.

Note that for order relations one only needs lemmata for one direction as the other can be derived from them. For example, one does not require both the lemma $\forall a, b, c.\ a \le b \Rightarrow b \le c \Rightarrow a \le c$ and $\forall a, b, c.\ a \ge b \Rightarrow b \ge c \Rightarrow a \ge c$.

If NUPRL finds a lemma missing in the course of constructing a rewrite justification it prints out an error message suggesting the kind and structure of the missing lemma. After adding an appropriate lemma to the library, you need to evaluate the function `initialize_rw_lemma_caches ()` to make it accessible to the rewrite package.

8.9.1.5 Applying Conversions

The following tactics can be used to make conversions applicable to some clause of a proof goal.

`Rewrite` c i

Apply conversion c to clause i. The subgoal with the result of the conversion is labelled `main` while the labels of the other subgoals fall into the `aux` class.

A shorthand notation for `Rewrite` is `RW`. The following variants of `Rewrite` provide a controlled application of the conversion c to the subterms of clause i by combining `Rewrite` with conversionals (see Section 8.9.4) in various fashions.

`RWH` c i ≡ `RW (HigherC` c`)` i :
Apply c to the first possible subterm of clause i, starting from the root.

`RWU` c i ≡ `RW (SweepUpC` c`)` i :
Apply c to all subterms of clause i, starting from the leaves.

`RWD` c i ≡ `RW (SweepDnC` c`)` i :
Apply c to all subterms of clause i, starting from the root.

`RWN` n c i ≡ `RW (NthC` n c`)` i :
Apply c to the n-th immediate subterm of clause i.

`RWAddr` *addr* c i ≡ `RW (AddrC` *addr* c`)` i :
Apply c to the subterm of clause i whose address in *addr*

`RewriteType` c i

Apply conversion c to the type of a member or equality term in clause i.
The advantage of this tactic over `Rewrite` is that this generates simpler well-formedness goals. In particular, it generates no well-formedness goals involving the equands of the equality or the element of the member term. A shorthand notation for `RewriteType` is `RWT`

For testing the effect of a conversion c on a term t with an empty environment one may evaluate the expression `apply_conv` c t in the refiner top loop.

8.9.1.6 Conversion Arguments

The descriptions of conversions in the sections below assume that the conversions have been applied to an environment e and a term t. Types of arguments to conversions are:

$c*$	:	`convn`	*type of conversions*
$e*$	:	`env`	*environment*
i j	:	`int`	*hypothesis or clause indices*
addr	:	`int list`	*subterm address*
a	:	`tok`	*name of abstraction*
name	:	`tok`	*name of lemma or cached conversion*
$t*$	:	`term`	

A suffix s on the name of an argument indicates that it is a list. For example cs is considered to have type `conv list`.

8.9.2 Atomic Conversions

Atomic conversions are the basic building blocks for constructing conversions. They may rewrite terms according to given lemmata and hypotheses, fold and unfold abstractions, or evaluate primitive and abstract redices. For the sake of completeness, there are also two trivial conversions.

`IdC`
: The identity conversion, which does not change a term

`FailC`
: The conversion that always fails

8.9.2.1 Lemma and Hypothesis Conversions

Lemma and hypothesis conversions derive rewrite rules from lemmata and hypotheses that contain either simple or general universal formulae (see Section 8.2.5). The consequents of these formulae must be of form $a\ r\ b$, where r is a relation as described in Section 8.9.1.2.

Usually we describe these conversions as rewriting in a left-to-right direction: they replace instances of a's by instances of b's. Each conversion also has a twin conversion that works right-to-left, which is indicated by a prefix `Rev` to their names.

`LemmaC` *name*
`RevLemmaC` *name*
: If lemma *name* contains a simple universal formula with consequent $a\ r\ b$, rewrite instances of a to instances of b (or instances of b to instances of a, respectively).

`HypC` *i*
`RevHypC` *i*
: If hypothesis *i* contains a simple universal formula with consequent $a\ r\ b$, rewrite instances of a to instances of b (or instances of b to instances of a, respectively).

The above conversions are instances of two more general conversions

```
GenLemmaWithThenLC (n:int) (hints:(var#term) list) (Tacs:tactic list) (name:tok)
GenHypWithThenLC   (n:int) (hints:(var#term) list) (Tacs:tactic list) (i:int)
```

which rewrite according to *general* universal formulae in lemmata and hypotheses. The meaning of their arguments is as follows:

n indicates the *n*-th consequent of a general universal formula. If `-1` is used then the formula is always treated as simple. In particular a ⇔ relation will be considered the relation in the consequent rather than a part of the structure of the general universal formula.

hints supply bindings for variables in the formula that NUPRL's matching routines cannot guess.

Tacs is used for *conditional* rewriting, i.e. when the antecedents of a formula have to be checked for validity before the rewrite rule is used.
The tactics in *Tacs* are paired up with the subgoals formed from instantiated antecedents. The rewrite goes through only if each tactic completely proves its corresponding subgoal. If there are fewer tactics than antecedents, an appropriate number of copies of the head of *Tacs* will be added to left of *Tacs*. If *Tacs* is empty, then rewriting must go through unconditionally.

name is the name of the lemma.

i is the number of a hypothesis. Negative numbers are allowed (c.f. Section 8.2.1.1).

Other useful specializations of GenLemmaWithThenLC and GenHypWithThenLC are:

GenLemmaC *n name*	≡	GenLemmaWithThenLC *n* [] [] *name*
LemmaWithC *hints name*	≡	GenLemmaWithThenLC (-1) *hints* [] *name*
LemmaThenLC *Tacs name*	≡	GenLemmaWithThenLC (-1) [] *Tacs name*
GenHypC *n i*	≡	GenHypWithThenLC *n* [] [] *i*
HypWithC *hints i*	≡	GenHypWithThenLC (-1) *hints* [] *i*
HypThenLC *Tacs i*	≡	GenHypWithThenLC (-1) [] *Tacs i*

The hypothesis conversions described here derive their rewrite rules from *local* environments (Section 8.9.1.1) that they are presented with on their first applications. If the conversions are applied with conversionals such as HigherC or NthC that start applying a conversion at the top of a term, then the environment is always the same as the environment of the clause being rewritten.

8.9.2.2 Atomic Direct-Computation Conversions

Low level direct-computation conversions are not usually invoked directly by the user, but useful for controlling the evaluation of redices and the folding/unfolding of abstractions in advanced rewrite strategies. Computation conversions for interactive invocation are described in Section 8.9.3.

UnfoldTopAbC
UnfoldsTopC *as*
UnfoldTopC *a* ≡ UnfoldsTopC [*a*]

Unfold t if it is an abstraction (with operator identifier listed in *as*).

AUnfoldsTopC *attrs*

Unfold t if it is an abstraction that has any of the attributes in *attrs* (see Section 7.1.3).

RecUnfoldTopC *a*

Unfold the recursive definition of *a* if it occurs on the top level of t.

FoldsTopC *as*
FoldTopC *a* ≡ FoldsTopC [*a*]

Try to fold an instance of an abstraction whose operator identifier is listed in *as*

RecFoldTopC *a*

Try to fold an instance of the recursively defined term *a* on the top level of t.

RecUnfoldTopC and RecFoldTopC work only with recursive definitions that have been introduced with the AddRecDef* button (see Section 4.3.2.2).

TagC *tagger*

Do forward computations on the term t as indicated by the tags in (*tagger* t).

RedexC

Contract t if it is a primitive redex.

AbRedexC
ForceRedexC *force*

Contract t if it is a primitive or abstract redex (of strength less or equal to *force*).

AnyExtractC
ExtractC *names*

Expand t if it is an extract term (of a theorem listed in *names*).

Abstract redices. Primitive redices that are buried under abstractions are called *abstract redices*. For example, the first and second projection functions for pairs are abstractions:

```
*A pi1     t.1  ≡  let <x,y> = t in x
*A pi2     t.2  ≡  let <x,y> = t in y
```

and the term ⌞<a,b>.1⌟ is an abstract redex, which contracts to the term ⌞a⌟.

To contract abstract redices, the conversion AbRedexC consults an abstract redex table, whose entries are created using the function

add_AbReduce_conv *opid* *c*

where *opid* is the operator identifier of the outermost term of the redex[10] and *c* is a conversion for contracting instances of the redex. Instances of add_AbReduce_conv are usually included in ML objects positioned immediately after the definitions of non-canonical abstractions.

An alternative method for indicating an abstract redex is to associate a 'reducible' attribute (see Section 7.1.3) with a non-canonical abstraction using the function

add_reducible_ab *opid*

The AbRedexC conversion first unfolds all reducible abstractions at the top level of the term before further analyzing it to see if it is a redex. When this method is applicable, it is more concise than using add_AbReduce_conv.

Reduction Strengths and Forces. In some situations it is desirable to have some redices contracted but not others. To this end, one may specify the *strength* of a redex and provide an optional *force* argument to tactics invoking direct computation conversions. Strengths and forces are arranged in a partial order on ML tokens. A redex is contracted only if the reduction force applied to it is greater or equal to its strength. A strength is associated with a redex in two ways.

- The strength is directly associated with the reduction rule for the redex.
- The strength is associated with the canonical term that is the principal argument of the redex. Strengths can be associated with canonical terms using the function

  note_reduction_strength *opid* *strength*

The currently supported strengths, in increasing order, are

‘1‘	beta redices
‘2‘	other primitive redices
‘3‘	abstract redices recursive
‘4‘	abstract redices non-recursive
‘6‘	module projection functions with coercion arguments
‘7‘	functions creating module elements from concrete parts.
‘8‘	quasi-canonical redices.
‘9‘	irreducible terms.

Contraction conversions that should be sensitive to the force of reduction can be added to the abstract redex table using the function

add_ForceReduce_conv *opid* *c*

where *opid* is the operator identifier of the outermost term of the redex and *c* is a conversion that takes a token argument for the force with which contraction of the redex is being attempted.

[10] If the outermost term is an (iterated) apply term, then *opid* refers to the operator identifier of the term at the head of the application.

8.9.3 Composite Direct Computation Conversions

The following conversions are commonly used for folding and unfolding abstractions and for the evaluation of primitive and abstract redices in a term.

`UnfoldsC` *as*
`UnfoldC` *a*
  Unfold all occurrences of abstractions listed in *as*, starting at the leaves of t.

`RecUnfoldC` *a*
  Unfold all occurrences of the recursive definition of *a* in t.

`FoldsC` *as*
`FoldC` *a*
  Fold all instances of abstractions whose operator identifiers are listed in *as*.

`RecFoldC` *a*
  Fold all instances of the recursively defined term *a* in t.

`ReduceC` ≡ `AbReduceC` ≡ `PrimReduceC`
`ForceReduceC` *force*
  Repeatedly contract all redices in t (with maximal strength *force*) starting from the root.

`EvalC` *as*
  Repeatedly unfold abstractions listed in *as* starting at the leaves, and then contract all redices.

`NormalizeC`
`SemiNormC` *as*
  Repeatedly unfold abstractions (listed in *as*), starting at the root, and then contract redices.

`IntSimpC`
  Rewrite t into arithmetical canonical form.

8.9.4 Conversionals

Conversionals provide the means for sequencing conversions in a way similar to the basic tacticals described in Section 8.8.1 and to apply them to subterms in a controlled fashion.

c_1 `ANDTHENC` c_2
  Apply c_2 to the result of c_1. Fail if either c_1 or c_2 fails.

c_1 `ORTHENC` c_2
  Apply c_2 to the result of c_1, or to t if c_1 fails

c_1 `ORELSEC` c_2
  Apply c_1. If this fails, apply c_2.

`RepeatC` c
`RepeatForC` n c
  Repeat c until it fails (exactly n times).

`ProgressC` c
  Apply c, but fail if c does not change the term.

`TryC` c
  Apply c. If this fails, leave the term unchanged.

`AllC` *cs*
  Iteratively apply all conversions from *cs*. Fail if one of them fails.

`SomeC` *cs*
  Iteratively apply all applicable conversions from *cs*.

`FirstC` *cs*

Apply the first applicable conversion from *cs*.

`SubC` c

Apply c to all immediate subterms of t (in left-to-right order).

`SubIfC` p c

Apply c to all immediate subterms of t that satisfy the proposition p

`NthSubC` n c

Apply c to the n-th immediate subterm of t.

`AddrC` *addr* c

Apply c to the subterm of t with term address *addr*.

`HigherC` c

`LowerC` c

Apply c the first applicable subterm of t, starting from the root (leaves).

`NthC` n c

Execute the n-th successful application of c to the subterms of t, starting from the root.

`SweepDnC` c

`SweepUpC` c

Apply c to all subterms of t, starting from the root (leaves).

`TopC` c

`DepthC` c

Repeatedly apply c to the first applicable subterm of t, starting from the root (leaves).

8.9.5 Macro Conversions

Macro conversions allow to express rewrite rules via pattern terms that describe the left and the right hand side of a transformation.

`MacroC` *name* c_1 t_1 c_2 t_2

Rewrite an instance of t_1 to the corresponding instance of t_2 using forward and reverse computation steps.

c_1 and c_2 must be direct computation conversions that rewrite the pattern terms t_1 and t_2 to the same term. `MacroC` uses second-order matching when matching instance terms against t_1. *name* is a failure token, which will be returned if rewriting fails.

`SimpleMacroC` *name* t_1 t_2 *as*

Rewrite an instance of t_1 to an instance of t_2 by unfolding abstractions from *as* and contracting redices.

`FwdMacroC` *name* c t

Rewrite an instance of t to an instance of the term resulting from applying c to t.

Using macro conversions enables a user to express rewrite steps in a user-defined theory in a way that does not reveal the underlying type theory. An example for the use of `MacroC` is the conversion for unrolling the Y combinator, defined in the theory `core_2`:

```
*A ycomb          Y ==  λf.(λx.f (x x)) (λx.f (x x))
*M ycomb_unroll  let YUnrollC =
                    MacroC ‘YUnrollC‘
                     (AllC [UnfoldC ‘ycomb‘; RedexC; RedexC])    ⌜Y F⌝
                     (AllC [UnfoldC ‘ycomb‘; AddrC [2] RedexC])  ⌜F (Y F)⌝
                 ;;
```

For another example, look at the `length_unroll` object in the `list_1` theory. `SimpleMacroC` can be used if the rewrite steps involved in justifying the equivalence of t_1 and t_2 are simpler, as in the case of the abstract redex for the projection functions on products

```
let pi1_evalC =
  SimpleMacroC ‘pi1_evalC‘  ⌜<a, b>.1⌝  ⌜a⌝  [‘pi1‘]
;;
```