A Logic Level Specification of the NQTHM Simplification Process

Alessandro Coglio
Università di Genova
tokamak@mrg.dist.unige.it

Fausto Giunchiglia
IRST and Università di Trento
fausto@irst.it

Paolo Pecchiari
IRST
peck@irst.it

Carolyn Talcott
Stanford University
clt@sail.stanford.edu

Abstract: Our ultimate goal is to provide a framework and a methodology which will allow users to construct and extend complex reasoning systems by composing existing modules in a "plug and play" manner. To this end in a previous paper we have defined a general architecture for a class of reasoning modules and systems called *Open Mechanized Reasoning Systems* (*OMRSs*). An OMRS has three components: a *logic* component which consists of the assertions manipulated by the OMRS and the elementary deductions upon them; a *control* component which consists of the inference strategies of the OMRS; an *interaction* component which provides the OMRS with the capability of interacting with other systems, including OMRSs and human users. We have already developed a theoretical framework underlying the logic component of this architecture. In this paper we show how this formalism can be used in practice to specify the logic component of the simplification process of NQTHM, the Boyer-Moore theorem prover.

Contents

1. Introduction

Our ultimate goal is to provide a framework and a methodology which will allow users to construct and extend complex reasoning systems by composing existing modules in a "plug and play" manner. In [7] a general architecture is proposed for a class of reasoning modules and systems called *Open Mechanized Reasoning Systems* (*OMRSs*). An OMRS has three components: a *logic* component which consists of the assertions manipulated by the OMRS and the elementary deductions upon them; a *control* component which consists of the inference strategies of the OMRS; an *interaction* component which provides the OMRS with the capability of interacting with other systems, including OMRSs and human users. That work is focused on the development of the theory underlying the logic component.

This paper follows on the work described in [7]. Here we show how the theory and methodology introduced in [7] can be used in practice to specify the logic component of the simplification process of the Boyer-Moore theorem prover, NQTHM [5, 4]. We stress that the choice of NQTHM is not by chance. First of all a better understanding of the internal workings of this system is an interesting exercise per se. NQTHM is a sophisticated prover with complex reasoning heuristics, data structures and algorithms designed for efficient representation and processing of various sorts of information, and coded for optimal performance. This system has been used to mechanically prove some quite deep theorems from traditional mathematics, e.g. the Church-Rosser theorem for lambda calculus and Goedel's incompleteness theorem, and more "mundane" theorems, e.g. the correctness of a simple real-time control algorithm and the correctness properties of a microprocessor that has been fabricated (cf. [2, 8] for a rather lengthy list of applications of the system with references). Second it is important to remark that, as described in [3], the interoperation among the modules inside NQTHM is very complex and cannot be described as a combination of black boxes. The formalization of the simplification process described in this paper is one of the most challenging tasks we could think of.

Plan: §2 and §3 contain background material. The NQTHM theorem prover and its simplification process are briefly described in §2. This description makes this work more self-contained, but it is in no way complete. For further details the reader is urged to consult [4], [5] and [3]. §3 is a brief summary of the main theoretical concepts described in much more detail in [7]. In §4 we present the formalization in our framework of NQTHM as an OMRS. Finally §5 contains some concluding remarks and describes future work.

2. The NQTHM Theorem Prover

The logic of NQTHM is based on a quantifier-free first-order extension of propositional calculus with equality. A theory in this logic implicitly defines a classical first-order theory, i.e. a set of well-formed expressions and a set of axioms. The expressions of a theory are called *terms*: a term is either a variable symbol or a sequence consisting of a function symbol of arity n followed by n terms. Terms are used instead of formulae in the prover[1]: when a term t is used as a formula (for instance as a conjecture to be proved), it represents the formula $t \neq \mathtt{F}$ (where `F` is a distinguished term that denotes "falsity"). Axioms can be incrementally added to the current theory by applying the *shell principle* (to axiomatize new inductively defined data types such as numbers, graphs, etc.), by applying the *definition principle* (to axiomatize new total functions), or by introducing arbitrary formulae as axioms.

NQTHM interacts with the user by means of high-level events, consisting in applications of the shell principle, applications of the definition principle, introductions of formulae as arbitrary axioms, or proofs of formulae. At any time the system keeps track of the sequence of events having taken place so far; such a sequence is called a *history*. A history implicitly defines a theory in the logic of NQTHM. The axioms and lemmas (i.e. formulae which have been proved) contained in a history are tagged by information indicating which inference techniques can use such axioms and lemmas (e.g. some axioms and lemmas are tagged as *rewrite rules*, indicating that they can be used by the rewriting module of the simplifier).

The search for a proof in NQTHM is driven by heuristics, called *inference processes*, that integrate several proof techniques: simplification; elimination of destructors; cross-fertilization; generalization; elimination of irrelevance; and induction. These processes get a formula in clausal form as input and return a set of clauses (considered conjunctively). A *clause* in the prover is a list of terms $l_1, \ldots, l_n$ (called *literals*), and represents the formula

$$l_1 \neq \mathtt{F} \vee \cdots \vee l_n \neq \mathtt{F}.$$

From a logical point of view all the processes are derived rules of inference that "run backwards" (cf. [5] p. 129). If a process returns the singleton set containing the input clause we say that the process *fails* (because it was not capable of "transforming" the input clause in any way), otherwise we say that it *succeeds*.

The top-level control of NQTHM (cf. [4] Chapter V) can be described in analogy to an (Escher) waterfall (see Figure 1). The user supplies a term which constitutes the conjecture to be proved. Such a term is first *preprocessed*: some non-recursive function definitions (called *abbreviations*) are expanded, and the resulting term is then converted to clausal form. The resulting clauses are put into the *top*. Repeatedly, a clause is removed from the top and *poured over the waterfall*, until none remains. A clause is poured over the waterfall by trying on it, in order, simplification, elimination of destructors, cross-fertilization, generalization, and elimination of irrelevance, until one succeeds. As soon as one succeeds, the returned clauses are put into the top. If none succeeds, the clause is put into the *pool*. When the top is empty the pool is *cleaned up*, i.e. some clauses subsumed by others are deleted. Then a clause is removed from the pool and fed into the induction process: if it

[1] This is made possible by means of axioms defining functional analogues of propositional connectives and equality.

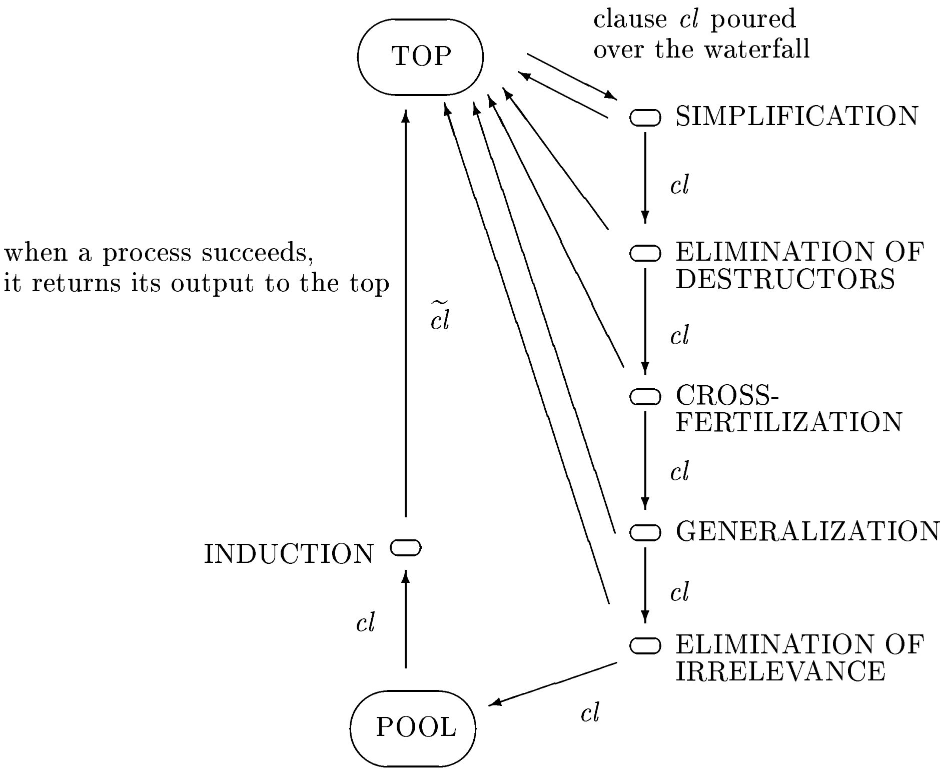
TOP
clause *cl* poured over the waterfall
SIMPLIFICATION
cl
when a process succeeds, it returns its output to the top
ELIMINATION OF DESTRUCTORS
$\widetilde{cl}$
cl
CROSS-FERTILIZATION
cl
GENERALIZATION
INDUCTION
cl
cl
ELIMINATION OF IRRELEVANCE
POOL
cl

Figure 1. The NQTHM waterfall

fails, then the entire proof fails, otherwise the resulting clauses are put into the top, from which a clause is then poured over the waterfall as explained above. When there is no clause left both in the top and in the pool, the initial conjecture has been proved.

However, there is one exception to the top-level algorithm sketched above. If the initial conjecture supplied by the user has been split in two or more clauses the first time the induction stage is reached, the induction process is tried on the initial conjecture and previous work on it is thrown away (see note on p. 90 of [4]). More precisely, the top and the pool are emptied, the initial conjecture is converted to clausal form (without expanding any abbreviation), and the resulting clauses are put into the pool; these clauses together are fed into the induction process[2], and then things proceed as explained in the algorithm above.

Since the 1979 version of NQTHM (described in [4]), various extensions of the prover have been built, including a linear arithmetic decision procedure [3], a goal-directed proof-checker interface, meta functions, function parameters, a limited form

[2] So, in fact the induction process receives a set of clauses, not just a clause, as input.

of quantification, and the use of hints for driving the heuristics. Apart from the linear arithmetic procedure, none of these features of the system is considered in this work.

2.1. The Simplifier

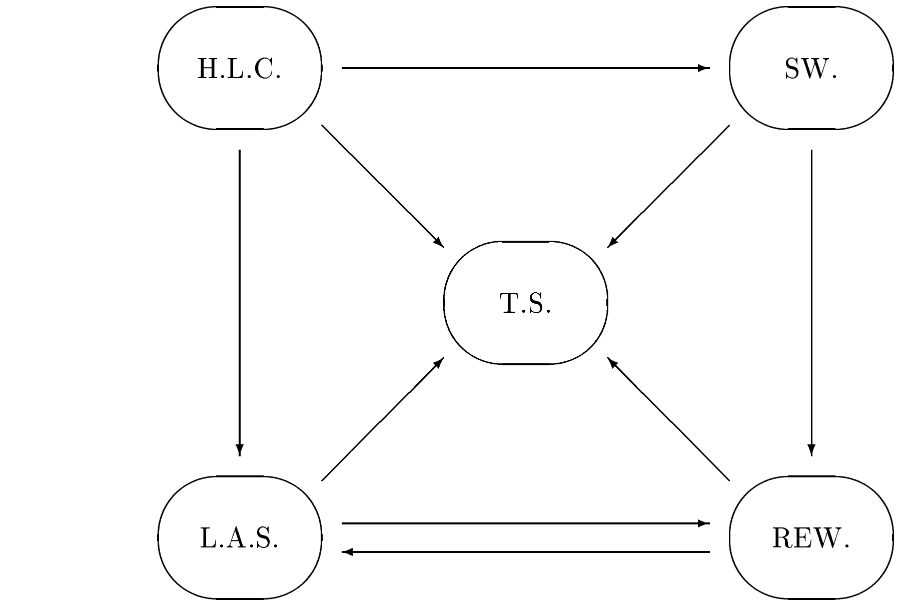

Figure 2. The modules of the NQTHM simplifier

The *simplifier* (i.e. the module in charge of the simplification process) consists of five component modules, depicted in Figure 2: typeset specialist (T.S.); rewriter (REW.); linear arithmetic specialist (L.A.S.); sweeper (SW.); and high-level controller (H.L.C.). Each arrow in Figure 2 indicates the call of a module by another module; these calls will be explained in the sequel.

2.1.1. The Typeset Specialist

As previously explained, new types are axiomatized by means of the shell principle. Thus, some terms denote objects of certain types: for instance, the term `(ADD1 (ADD1 (ADD1 (ZERO))))`[3] denotes the natural number[4] 3, whose type is called

[3] We use `this font` for NQTHM function symbols and variable symbols, and we employ the Boyer-Moore Lisp-like notation $(f\ t_1\ \ldots\ t_n)$ instead of a classical notation $f(t_1, \ldots, t_n)$.

[4] Natural numbers and other types are “built-in” in NQTHM, i.e. their axiomatization is present in any theory, as if the shell principle had been applied for each built-in type.

NUMBERP; the distinguished terms T and F respectively denote "truth" and "falsity", whose type is respectively called T and F. A *typeset* is just a set of types. We say that a term t *has typeset*, *ts*, iff under any interpretation of the variables of t, t denotes some object whose type is in *ts*. A *type alist* is a set of pairs, each pair consisting of a term and a typeset (with no distinct pairs containing a same term). A type alist is used to represent some assumptions, namely that each term has the associated typeset.

The typeset specialist can be used in two different ways. In the first way it is used to compute a typeset for a term, under the assumptions represented by a given type alist. In the second way it is used to assume a term true or false under the assumptions represented by a given type alist: the answer of the typeset specialist is that the term must be true under the type alist, or that the term must be false under the type alist, or that it may be either true or false; in the last case, two type alists are also returned, obtained by adding to the input type alist the assumption that the term is respectively true or false.

The two algorithms for computing typesets and assuming terms true or false are mutually recursive. We do not go into the details of their working. For more information about types and these algorithms, the reader is referred to [4] Chapter VI.

2.1.2. The Rewriter

The rewriter takes as input a term and a *context* (i.e. a collection of logical and control information, which we will partially describe below), and returns as output a term (obtained by rewriting the input term) and a set of terms (constituting *linearization hypotheses*; see next subsection). The context includes a *substitution*, i.e. a finite map from variables to terms: the term which is rewritten is in fact the result of applying the substitution to the input term. This substitution is used to avoid explicitly applying it in some recursive calls of the rewriter, but for simplicity we assume that it is explicitly applied and we ignore it in the sequel. The context also includes a *modality of equivalence*, i.e. a flag indicating if *identical* equivalence or *propositional* equivalence must be preserved during rewriting: the first means that two terms are equal (in the NQTHM logic), the second that one is equal to F iff the other is such. Propositional equivalence is used to rewrite literals of clauses, tests of IF[5], and hypotheses of rewrite rules (see below); in all other cases identical equivalence is used. The logical specification satisfied by the rewriter is that, under the logical assumptions contained in the context and under the assumption that the linearization hypotheses are true, the input term and the output term are equivalent according to the input modality of equivalence.

The algorithm used by the rewriter can be roughly explained as follows (cf. [4] Chapter IX and [3]). First, if the input term is a variable, or if it denotes a

[5] IF is a distinguished ternary function symbol whose axiomatization, defined as obvious, is built-in in NQTHM. Other function symbols whose axiomatizations are built-in in NQTHM, are EQUAL (i.e. the functional analogue of equality), NOT (i.e. the functional analogue of negation), and LESSP (i.e. the less-than relation over natural numbers).

constant of some type (e.g. a natural number), it is just returned. Otherwise, if the input term is an `IF`-expression (i.e. a term whose top function symbol is `IF`), we proceed as follows. The rewriter recursively calls itself to rewrite the test of the `IF` with propositional equivalence. It is now time to say that the input context of the rewriter includes a type alist. The typeset specialist is called to assume true or false the rewritten test of the `IF` under the type alist of the context: if it must be true or false, the left or right branch of the `IF` is recursively rewritten and returned; otherwise, the left and right branches are both recursively rewritten under contexts containing the appropriate one of the two type alists returned by the typeset specialist, thus obtaining an `IF`-expression with the rewritten arguments. Now the rewriter tries to *apply* one of the following *rewrite rules* (in this order) to the obtained `IF`-expression:

$$\texttt{(IF X Y Y)} = \texttt{Y};$$

$$\texttt{(IF X X F)} = \texttt{X};$$

$$\texttt{(IF X T F)} = \texttt{X}.$$

Applying one such rule means finding a substitution for `X` (and, if present, `Y`) which makes the left term of the rule syntactically identical to our `IF`-expression, and in this case the `IF`-expression is replaced with the result of applying the substitution to the right term of the rule. However, the third rule is tried only if the test is *boolean* (i.e. its typeset computed by the typeset specialist, under the type alist of the context, is $\{\underline{\texttt{T}},\underline{\texttt{F}}\}$). The term obtained by the first applicable rule, if any, is then returned; if no rule is applicable, the `IF`-expression is returned.

Otherwise, if the input term is an `EQUAL`-expression, we proceed as follows. First the two arguments are recursively rewritten, thus obtaining a new `EQUAL`-expression. If the two rewritten arguments are syntactically identical, the rewriter returns `T`. Otherwise, some sufficient conditions (described in [4] p. 121) for the falsity of the equality of the two terms are tested (e.g. one of these conditions is that the typesets of the two terms have empty intersection): if at least one is verified, `F` is returned. Otherwise, the following rewrite rules are tried (the first is applied only if the first argument is boolean), analogously to `IF`-expressions:

$$\texttt{(EQUAL X T)} = \texttt{X};$$

$$\texttt{(EQUAL X (EQUAL Y Z))} = \texttt{(IF (EQUAL Y Z) (EQUAL X T) (EQUAL X F))};$$

$$\texttt{(EQUAL X F)} = \texttt{(IF X F T)}.$$

Let t be the result of the first applicable rule, or just the `EQUAL`-expression if no rule is applicable. At this point, the rewriter tries to apply each rewrite rule present in the current history (see below) to t, one after the other, until one is applicable: in this case the result of the application of the rewrite rule is recursively rewritten; if no rule has been applied, t is just returned.

Otherwise, if the input term is a *recognizer expression* (i.e. a term whose top function symbol is a *recognizer*[6], we proceed as follows. First the argument of the expression is recursively rewritten. The typeset specialist is then called to compute the typeset of the rewritten argument. If such a typeset only contains the type associated to the recognizer, `T` is returned; if it does not contain that type, `F` is returned. Otherwise, the rewriter tries to apply a rewrite rule in the current history (see below): if one is found, the result is recursively rewritten and then returned; otherwise the recognizer expression is just returned.

Otherwise, let the input term be $(f\ t_1\ \ldots\ t_n)$. First all the arguments are recursively rewritten, obtaining $t'_1, \ldots, t'_n$. If f has a definition in the current history (i.e. if f has been axiomatized by applying the definition principle) and if certain heuristic conditions are met, $(f\ t'_1\ \ldots\ t'_n)$ is *unfolded*, and the rewriter returns the result of recursively rewriting the body. From a logical point of view, a definition of f is a formula

$$(f\ v_1\ \ldots\ v_n) = \mathit{body}$$

where $v_1, \ldots, v_n$ are distinct variables and *body* is a term satisfying certain conditions; *unfolding* the term $(f\ t'_1\ \ldots\ t'_n)$ amounts to replacing it with the result of applying to *body* the substitution mapping each v_i to t'_i. If f has no definition or the heuristic conditions are not met, then we try to apply a rewrite rule of the current history (see below) to $(f\ t'_1\ \ldots\ t'_n)$. If one is found then the result is recursively rewritten and then returned by the rewriter. Otherwise, if f has a definition and if some heuristic conditions are met (different from those above), our term is unfolded and the rewriter returns the result of recursively rewriting the body; in case f has no definition or these heuristic conditions are not met, $(f\ t'_1\ \ldots\ t'_n)$ is just returned.

We now explain how a *rewrite rule* of a history is *applied* to a term t. From a logical point of view, a rewrite rule (axiom or lemma) represents a formula of two possible forms:

$$l_1 \wedge \cdots \wedge l_n \rightarrow (\mathit{lhs} = \mathit{rhs});$$

$$l_1 \wedge \cdots \wedge l_n \rightarrow (\mathit{lhs} \leftrightarrow \mathit{rhs}).$$

The terms $l_1, \ldots, l_n$, where $n \geq 0$, are called *hypotheses* of the rule; the formula $(\mathit{lhs} = \mathit{rhs})$ or $(\mathit{lhs} \leftrightarrow \mathit{rhs})$ is called *conclusion* of the rule. Rules of the second form may only be applied when t is rewritten preserving propositional equivalence. To apply a rule, first a substitution is searched for which when applied to t makes it syntactically identical to *lhs*; if no such substitution exists, the rule is not applicable. Otherwise, the hypotheses obtained by applying the substitution to $l_1, \ldots, l_n$ are recursively rewritten with propositional equivalence. If all of them rewrite to `T`, then the rule is applicable, and the result of its application is obtained by applying the substitution to *rhs*. To be more precise, if some hypothesis to be rewritten has

[6] Each type axiomatized in NQTHM has an associated unary function symbol called *recognizer* of the type, which “returns” `T` if the argument denotes an object of the type, `F` otherwise. For example, `NUMBERP` is the recognizer for the type of natural numbers.

NOT as top function symbol, then the argument is rewritten instead, and of course it must be rewritten to F.

For ease of exposition, we have omitted an extremely important feature of the algorithm of the rewriter, namely its call to the linear arithmetic specialist; we now fill this gap. It must be said that the context includes an *objective*, indicating (if the modality of equivalence is propositional) that the term must be rewritten to T or to F or to any of them; the objective has one of the first two values only when rewriting a hypothesis (or its negation if the hypothesis is a NOT-expression) of a rewrite rule. The call to the linear arithmetic specialist happens just before trying to apply a rewrite rule to the term, if the objective is T or F, and if the term or its negation (if the term is a NOT-expression) has EQUAL or LESSP as top function symbol. The term (if the objective is F) or its negation (if the objective is T) is fed into the linear arithmetic specialist to derive a contradiction (see next subsection). If no contradiction is derived, we proceed to try rewrite rules, as previously explained. If a contradiction is derived, a set of linearization hypotheses (see next subsection) is returned to the rewriter, representing assumptions under which the contradiction holds: in this case the rewriter returns T or F (according to the objective), together with the linearization hypotheses. The rewriter collects all the linearization hypotheses returned by its calls to the linear arithmetic specialist, and returns them together with the rewritten term.

Finally, it must be said that when the rewriter returns a term, it first checks if a typeset is associated to the term in the type alist of the context. If no typeset is found, the term is returned as it is. Otherwise, if the typeset is $\{\underline{\texttt{T}}\}$ or $\{\underline{\texttt{F}}\}$ then respectively T or F is returned instead; if the typeset does not include $\underline{\texttt{F}}$ and the modality of equivalence is propositional, T is returned. Otherwise the term is returned as it is.

2.1.3. The Linear Arithmetic Specialist

The linear arithmetic specialist implements a (partial) decider for linear inequalities over the natural numbers, which works by refutation and is based on a simple rational-based procedure that formalizes the “high school idea of cross multiplying and adding equalities to eliminate variables” (see [3] p. 10). A linear inequality, called a *polynomial*, has the form

$$i + i_1 * t_1 + \cdots + i_n * t_n \leq 0$$

where i is an integer called the *constant*, $i_1, \ldots, i_n$ are integers called the *coefficients*, and $t_1, \ldots, t_n$ are terms called the *multiplicands*. A polynomial is called *vacuous* iff $i \leq 0$ and $i_j \leq 0$ for each $1 \leq j \leq n$ (because it is trivially true); when a vacuous polynomial is derived, it is discarded. A polynomial is called *impossible* iff $i > 0$ and $i_j \geq 0$ for each $1 \leq j \leq n$ (because it is trivially false); when an impossible polynomial is derived, a contradiction has been found.

The linear arithmetic specialist takes as input a type alist, a *polynomial database* (i.e. a set of polynomials), and a set of terms, and returns as output a new polynomial database. A database represents a set of assumptions, namely that all the polynomials it contains are true. The output database is obtained by adding to the input one the polynomial assumptions which can be derived by assuming all the input terms true. The algorithm can be roughly described as follows: first, the input terms are *linearized* (i.e. "converted" to polynomials), and the resulting polynomials are *pushed* into the input database; then the database is *augmented with linear rules.* We now explain in detail these steps.

Pushing some polynomials into a database consists in inserting the polynomials into the database, then performing all the *admissible* (according to certain heuristic conditions) cross-multiplications, thus generating further polynomials. Furthermore, each polynomial containing a multiplicand t with a positive coefficient and satisfying some heuristic conditions, is cross-multiplied with the *non-negative assumption* for t, i.e. the polynomial

$$0 - 1 * t \leq 0$$

(cf. [3] p. 45).

A history includes *linear rules*, which from a logical point of view are formulae of two possible forms:

$$l_1 \wedge \cdots \wedge l_n \rightarrow (\texttt{LESSP}\ lhs\ \ rhs);$$

$$l_1 \wedge \cdots \wedge l_n \rightarrow (\texttt{NOT}\ (\texttt{LESSP}\ lhs\ \ rhs)).$$

The terms $l_1, \ldots, l_n$, where $n \geq 0$, are called the *hypotheses* of the rule; the term (LESSP *lhs rhs*) or (NOT (LESSP *lhs rhs*)) is called the *conclusion* of the rule. *Augmenting* a database *with linear rules* consists in the following. All the linear rules of the current history are examined, one after the other, to check if they satisfy certain heuristic conditions. For each linear rule satisfying the heuristic conditions, the results of applying a substitution, which is determined during the satisfaction check of the conditions, to the hypotheses $l_1, \ldots, l_n$ are fed into the rewriter (together with a context containing the input type alist of the linear arithmetic specialist, and the propositional modality of equivalence). If all of them rewrite to T, then the results of applying the substitution to *lhs* and *rhs* are fed into the rewriter obtaining lhs' and rhs'; then (LESSP lhs' rhs') or (NOT (LESSP lhs' rhs')) is linearized and the resulting polynomials pushed into the database.

We do not go into the details of how terms are *linearized* to polynomials (see [3] for further information). The important thing is that the linearization of a term produces a set, whose cardinality is 2 or less, of sets of polynomials which derive from assuming the term true: such a set represents a disjunction (across the set of sets) of conjunctions (across the sets) of polynomials. If the set of sets is empty, then no polynomial can be derived from the term. If it is a singleton set (i.e. we just obtain a conjunction of polynomials), the resulting polynomials can be pushed into the database. If it is a doubleton set, we have a disjunction of two conjunctions, and each conjunction is called an *alternative.* In this case, since a disjunction cannot be pushed into the database, the linear arithmetic specialist proceeds as follows.

The database is "saved", and the first alternative is pushed into the database, which is then augmented with linear rules: if an impossible polynomial is derived in this way, the saved database is restored and the second alternative can be pushed into the database. If no impossible polynomial is found, the symmetric procedure is performed, swapping first and second alternative. If again no impossible polynomial is found, the pair of alternatives is just discarded and no polynomial is pushed into the database.[7]

For ease of exposition, in the above explanations we have not spoken about linearization hypotheses. When a polynomial is produced by linearizing a literal, also a set of terms, called *linearization hypotheses*, is produced, which constitute conditions under which the polynomial derives from the term[8]. In fact each polynomial has attached to it a set of linearization hypotheses, so that a polynomial really represents an assertion

$$lh_1 \wedge \cdots \wedge lh_n \rightarrow i + i_1 * t_1 + \cdots + i_n * t_n \leq 0$$

(where $lh_1, \ldots, lh_n$, with $n \geq 0$, are the linearization hypotheses). So, when two polynomials are cross-multiplied, the union of the two sets of linearization hypotheses is attached to the resulting polynomial. Furthermore, in case pushing one of two alternatives into the database produces an impossible polynomial, the linearization hypotheses of the impossible polynomial are added to those of all the polynomials of the other alternative, before pushing it into the database.

Finally, something must be said which also completes the description of the rewriter we gave in the previous subsection. The input context of the rewriter also includes a polynomial database. When the rewriter calls the linear arithmetic specialist to derive a contradiction, the database of the context is fed into the linear arithmetic specialist; moreover, when this module calls the rewriter (e.g. to rewrite the hypotheses of a linear rule to T) the current database is fed into the rewriter. Of course, the linearization hypotheses returned by the rewriter consist of all the linearization hypotheses of the impossible polynomials found when contradictions are derived by the calls to the linear arithmetic specialist.

[7] The arrow from the linear arithmetic specialist to the typeset specialist in Figure 2 refers to calls performed during linearization.

[8] For instance, the linearization of the term (LESSP I (SUB J I)), where SUB denotes subtraction, produces the polynomial $1+(-1)*\mathtt{J}+2*\mathtt{I} \leq 0$, and the linearization hypothesis (NOT (LESSP J I)). The reason for such a linearization hypothesis is that in NQTHM the axiomatization of natural numbers coerces to 0 all the terms not representing natural numbers (e.g. (SUB J I) "evaluates" to 0 if the value "assigned" to I is greater than the value "assigned" to J). See [3] for further information.

2.1.4. The Sweeper

The sweeper takes as input a clause and a polynomial database (which, as we will see in the next subsection, is obtained by assuming the clause false), and returns as output a set of clauses whose conjunction implies the input clause.

Before describing the algorithm of this module, we must define a couple of concepts. Given a clause cl and some literals $l_1, \ldots, l_n$, the result of *splitting* cl on $l_1, \ldots, l_n$ (cf. [3] p. 55) is the set of n clauses obtained by adding each of $l_1, \ldots, l_n$ to cl. Given some clauses $cl_1, \ldots, cl_n$ and a clause cl containing a literal l, the result of *splicing* $cl_1, \ldots, cl_n$ in place of l into cl (cf. [3] p. 56), is the set of n clauses obtained by replacing l in cl with the literals of each of $cl_1, \ldots, cl_n$.

The sweeper performs the following procedure on each literal l of a clause cl, one after the other. Let $cl = cl' \vee l$. First the sweeper calls the typeset specialist to assume false all the other literals of the clause (i.e. the literals in cl'). If some of them must be true, a contradiction has been derived, and so the clause can be eliminated, because trivially true. Otherwise, the type alist, together with the database, the propositional modality of equivalence, and the literal itself, are fed into the rewriter, which returns a literal l' and some linearization hypotheses $lh_1, \ldots, lh_n$. From a logical point of view, this means that

$$lh_1 \wedge \cdots \wedge lh_n \rightarrow (cl \leftrightarrow cl' \vee l').$$

If l' is `T`, cl is true under the linearization hypotheses, so that we must prove it under the negation of each linearization hypothesis, i.e. we must prove

$$(\neg lh_1 \rightarrow cl) \wedge \cdots \wedge (\neg lh_n \rightarrow cl);$$

therefore cl is split on $lh_1, \ldots, lh_n$; the sweeper continues to perform the procedure on the literals of all the obtained clauses. If l' is `F`, l can be eliminated from cl under the linearization hypotheses, so that we must prove

$$lh_1 \wedge \cdots \wedge lh_n \rightarrow cl',$$

as well as cl under the negation of each linearization hypothesis; therefore l is replaced in cl with the negations of $lh_1, \ldots, lh_n$ and cl is also split on $lh_1, \ldots, lh_n$; the sweeper continues to perform the procedure on the literals of all these clauses. If l' is neither `T` nor `F`, l' is transformed into clausal form according to the `IF`-expressions present in it[9], obtaining $cl_1, \ldots, cl_m$ such that

$$l' \leftrightarrow cl_1 \wedge \cdots \wedge cl_m,$$

so that we must prove

$$lh_1 \wedge \cdots \wedge lh_n \rightarrow (cl' \vee cl_1) \wedge \cdots \wedge (cl' \vee cl_m),$$

[9] For instance, a term (`FN (IF X Y Z)`) is transformed into ((`NOT X`) $\vee$ (`FN Y`)) $\wedge$ (`X` $\vee$ (`FN Z`)). For details, see [4] Chapter IX.

as well as *cl* under the negation of each linearization hypothesis; therefore the negations of $lh_1, \ldots, lh_n$ are all added to each of $cl_1, \ldots, cl_m$, and these clauses are then spliced in place of l into *cl*; furthermore *cl* is split on $lh_1, \ldots, lh_n$; the sweeper continues to perform the same procedure on the literals of all the obtained clauses.

When the above procedure has been performed on all the literals of all the clauses, the final set of clauses is returned.

2.1.5. The High-Level Controller

Before invoking any other module, the high-level controller eliminates all the *trivial equations* from the clause given in input to the simplifier. A trivial equation is a literal of three possible forms

$$(\texttt{NOT (EQUAL } v\ t\texttt{))},$$

$$(\texttt{NOT (EQUAL } t\ v\texttt{))},$$

$$v,$$

where t is a term and v a variable not occurring in t. *Eliminating* a trivial equation of one of the two first forms consists in removing it from the clause and applying to the other literals the substitution which maps v to t. Eliminating a trivial equation of the third form consists in removing it from the clause and applying to the other literals the substitution which maps v to `F` (because a formula v is equivalent to the formula (`NOT` (`EQUAL` v `F`))).

After that, the typeset specialist is repeatedly called to assume false all the literals of the clause. If any of them must be true, the clause is trivially true and hence the simplifier returns the empty set of clauses.

Otherwise, the obtained type alist, together with the empty database and all the literals of the clause, are fed into the linear arithmetic specialist. If an impossible polynomial is found, then the clause is split on the linearization hypotheses of the impossible polynomial, and the resulting clauses are returned by the simplifier.[10]

Otherwise, the obtained database is searched for any two *mated* polynomials on a term t whose *conglomerated* term is t', which roughly means that the two polynomials imply "$t \leq t' \wedge t' \leq t$" (cf. [3] p. 53–54). If some heuristic conditions are met (see [3] p. 56), the clause is split on the union of the linearization hypotheses of the two mated polynomials; furthermore the negations of these linearization hypotheses, together with the negations of (`NUMBERP` t), (`NUMBERP` t') and (`EQUAL` t t'), are added to the clause itself. All the resulting clauses are returned by the simplifier.

[10] Actually, this procedure consists of two phases. In the first phase, so to say, only "part" of the assumptions of the type alist are made "available" to the linear arithmetic specialist, in order to accurately track which literals produced which polynomial and thus avoid tail biting. If a contradiction is found, the simplifier returns the result of splitting the clause on the linearization hypotheses. Otherwise, the second phase begins: the database is "saved", and it is fed into the linear arithmetic specialist together with all the assumptions of the type alist. If a contradiction is found, we proceed as above. Otherwise, the old database is restored, and we proceed as explained in the sequel.

If no mated polynomials are found, then the database and the clause are fed into the sweeper, and the returned set of clauses is also returned by the simplifier.

3. An Overview of Our Formalism

We start by introducing our notation conventions. Let Y, Y_0, Y_1, Y_2 be sets. We specify meta-variable conventions in the form: let y range over Y, which should be read as: the meta-variable y and decorated variants such as y', y_0, $\ldots$, range over the set Y. $Y_0 \times Y_1$ is the set of pairs with first component from Y_0 and second component from Y_1. If $\langle y_1, \ldots, y_n \rangle$ is an n-tuple, we use the notation $\langle y_1, \ldots, y_n \rangle \downarrow i$ to denote the i-th element y_i. Y^* is the set of finite sequences of elements of Y. We write $[y_1, \ldots, y_n]$ for the sequence of length n with i-th element y_i (thus $[\,]$ is the empty sequence). $u \diamond v$ denotes the concatenation of the sequences u and v. $\mathrm{P}_\omega(Y)$ is the set of finite subsets of Y. The empty set is denoted by $\emptyset$. We use the convention that if y ranges over Y, then $\overline{y}$ ranges over Y^* and $\widetilde{y}$ ranges over $\mathrm{P}_\omega(Y)$. $[Y_0 \xrightarrow{\mathrm{f}} Y_1]$ is the set of finite maps from Y_0 to Y_1. We use $\vec{\emptyset}$ to denote the (unique) finite map with empty domain. $[Y_0 \to Y_1]$ is the set of total functions, f, with domain, $\mathrm{Dom}(f)$, Y_0 and range, $\mathrm{Rng}(f)$, contained in Y_1. If $f \in [Y_0 \to Y_1]$ and $g \in [Y_1 \to Y_2]$, then $g \circ f \in [Y_0 \to Y_2]$ is the composition of f and g: $(g \circ f) = \lambda y.g(f(y))$. For any function f, $f\{y \mapsto y'\}$ is the function f' such that $\mathrm{Dom}(f') = \mathrm{Dom}(f) \cup \{y\}$, $f'(y) = y'$, and $f'(z) = f(z)$ for $z \neq y, z \in \mathrm{Dom}(f)$.

3.1. Sequent Systems, Rules, and Reasoning Theories

A reasoning theory consists of a sequent system and some rules. The sequent system specifies a set of sequents (i.e. assertions or judgements for consideration), and the rules specify deductions upon these sequents. In addition, the sequent system also specifies a set of constraints which can constrain the application of rules. Both sequents and constraints may contain schematic parts, which can be filled in through instantiation maps, also specified by the sequent system. Reasoning theories can be glued together in order to specify, at the logical level, the integration of different systems.

A *sequent system* is a structure of the form

$$Ssys = \langle S, C, \models, I, \_\mathtt{[\_]} \rangle.$$

S is the set of *sequents.* C is the set of *constraints.* $\models \subseteq (\mathrm{P}_\omega(C) \times C)$, is a consequence relation on constraints, which abstractly represents a constraint solving (satisfaction) mechanism. I is the set of *instantiation maps* (or *instantiations*); there exists a binary composition operation $\circ$ over instantiations, and I is closed under composition. $\_\mathtt{[\_]}$ is the operation for application of instantiations to sequents and to constraints, that is $\_\mathtt{[\_]} : S \times I \to S$ and $\_\mathtt{[\_]} : C \times I \to C$. From now on, we let s range over S, c range over C, and ι range over I. In the following we describe the requirements that such a structure must meet in order to qualify as a sequent system.

Constraint satisfaction must obey the basic laws for a (classical) consequence relation, i.e. reflexivity, monotonicity, and cut (cf. [1, 9]). We extend satisfaction to a relation between sets of constraints by defining $\widetilde{c} \models \widetilde{c}_1$ iff $(\forall c \in \widetilde{c}_1)(\widetilde{c} \models c)$. Let us call *schematic entities* the entities that I acts on producing entities of the same sort. The collection of schematic entities includes sequents and constraints, and it is closed under formation of finite sets or sequences, and finite maps whose range is a set of schematic entities with instantiation extended pointwise. One of the most basic kinds of constraint is syntactic equality between schematic entities, used for example in matching and unification. We assume that equations $s \sim s'$ between sequents are among the constraints, and that instantiation propagates to the sequent terms, that is $(s \sim s')[\iota] = s[\iota] \sim s'[\iota]$. $\models$ obeys, w.r.t. equations between sequents, the usual laws for equality, i.e. reflexivity, transitivity, and symmetry. Instantiations preserve satisfaction, i.e. if $\widetilde{c} \models c$, then $\widetilde{c}[\iota] \models c[\iota]$.

Let $Ssys = \langle S, C, \models, I, \_[\_]\rangle$ be a sequent system, and let Id be a set of identifiers. Then the set of *rules*, $\mathbf{Rule}[Ssys]$, over $Ssys$, and the set of *rule sets*, $\mathbf{Rset}[Ssys, Id]$, over $(Ssys, Id)$ are defined by

$$\mathbf{Rule}[Ssys] = \{R \subseteq (S^* \times S \times \mathrm{P}_\omega(C)) \mid (\forall \langle \overline{s}, s, \widetilde{c}\rangle \in R)(\forall \iota \in I)(\langle \overline{s}, s, \widetilde{c}\rangle[\iota] \in R)\},$$

$$\mathbf{Rset}[Ssys, Id] = [Id \xrightarrow{\mathrm{f}} \mathbf{Rule}[Ssys]].$$

If $\widetilde{r} \in \mathbf{Rset}[Ssys, Id]$ and $id \in Id$ we say that $\langle \overline{s}, s, \widetilde{c}\rangle \in \widetilde{r}(id)$ is an *instance* of id with *premisses*, $\overline{s}$, *conclusion*, s, and *applicability conditions*, $\widetilde{c}$. We may write $\langle id, \overline{s}, s, \widetilde{c}\rangle \in \widetilde{r}$ for $\langle \overline{s}, s, \widetilde{c}\rangle \in \widetilde{r}(id)$, and say that $\langle id, \overline{s}, s, \widetilde{c}\rangle$ is an instance of $\widetilde{r}$. A *rule generator* is any subset rg of $S^* \times S \times \mathrm{P}_\omega(C)$. The rule generated by rg is the set $rg[I]$. An *n-ary rule* is a rule whose instances all have n premisses.

A *reasoning theory*, Rth, is a structure

$$Rth = \langle Ssys, Id, \widetilde{r}\rangle$$

such that $Ssys$ is a sequent system, Id is a set of identifiers, and $\widetilde{r} \in \mathbf{Rset}[Ssys, Id]$ is a rule set.

The different reasoning modules which constitute a complex system (for example NQTHM) are described, at the logical level, by different reasoning theories: the whole system is described by gluing together these reasoning theories using additional inference rules among sequents of different reasoning theories, thus specifying how deductions performed in the different component modules relate to each other.

Let $Rth_1 = \langle Ssys_1, Id_1, \widetilde{r}_1\rangle$, ..., $Rth_n = \langle Ssys_n, Id_n, \widetilde{r}_n\rangle$ be disjoint reasoning theories, with $Ssys_i = \langle S_i, C_i, \models_i, I_i, \_[\_]_i\rangle$ and $\widetilde{r}_i \in \mathbf{Rset}[Ssys_i, Id_i]$ for $1 \leq i \leq n$. By *disjointness* we mean that the families of sets S_i, C_i, I_i, and Id_i for $1 \leq i \leq n$ are each pairwise disjoint. Thus $S_i \cap S_j = \emptyset$, for $1 \leq i \neq j \leq n$, etc.

The (disjoint) *union*, $Ssys$, of the sequent systems $Ssys_i$ for $1 \leq i \leq n$ is defined by

$$Ssys = \bigcup_{1 \leq i \leq n} Ssys_i = \langle S, C, \models, I, \_[\_]\rangle$$

where $S = \bigcup_{1\leq i\leq n} S_i$, $C = \bigcup_{1\leq i\leq n} C_i$, $I = I_1 \times \ldots \times I_n$. $\models$, $\_\,[\_]$, and $\circ$ are defined as follows:

$$\begin{aligned}
&\widetilde{c} \models c \quad \text{iff} \quad \widetilde{c} \cap C_i \models_i c \quad \text{if} \quad c \in C_i,\\
&x\,[\iota] = x\,[\iota \downarrow i]_i \quad \text{if} \quad x \in S_i \cup C_i,\\
&\iota \circ \iota' = \langle \iota \downarrow 1 \circ \iota' \downarrow 1, \ldots, \iota \downarrow n \circ \iota' \downarrow n \rangle.
\end{aligned}$$

It is easy to check that all the requirements for a sequent system are satisfied.

Let $Id = \bigcup_{1\leq i\leq n} Id_i$, and let Id_B be a set of identifiers disjoint from Id. Let $\widetilde{r} = \bigcup_{1\leq i\leq n} \widetilde{r}_i$, i.e. $\widetilde{r}(id) = \widetilde{r}_i(id)$ if $id \in Id_i$, and let $\widetilde{r}_B \in \mathbf{Rset}[Ssys, Id_B]$ be a set of inference rules over the joined sequent system. The gluing of the Rth_i via $\widetilde{r}_B$ is defined by

$$Rth = glueRth([Rth_1, \ldots, Rth_n], Id_B, \widetilde{r}_B) = \langle Ssys, Id \cup Id_B, \widetilde{r} \cup \widetilde{r}_B \rangle.$$

We say that Rth is a *composite reasoning theory*, with *components* Rth_i, and *glue* $(Id_B, \widetilde{r}_B)$. The elements of the rule sets $\widetilde{r}_i$ are called the *internal rules* of Rth_i. The elements of $\widetilde{r}_B$ are called *bridge rules*.[11]

3.2. Reasoning Structures and Derivations

Reasoning structures represent proof fragments that occur during the construction of a proof. They provide two independent forms of flexibility: horizontal and vertical. Horizontal flexibility provides flexibility in mode of proof construction, reuse of proof fragments, and schematic reasoning. It comes from being able to stitch together fragments rather like a patchwork quilt and to incrementally refine schematic information. Vertical flexibility provides control over the level of immediately visible detail. It comes from nesting of reasoning structures and the ability to encapsulate a substructure into a nesting link, or open up a nesting link. Vertical flexibility is motivated by the need to organize large complex structures hierarchically, to be able to examine them at different levels of depth and detail, or to focus on meaningful substructures.

We let $Rth = \langle Ssys, Id, \widetilde{r} \rangle$ be an arbitrary but fixed reasoning theory. We let SN (*sequent nodes*) and LN (*link nodes*) be two disjoint countable sets, used to construct reasoning structures. We give the definition of reasoning structures in two steps. First we define basic reasoning structures. They provide the horizontal dimension of flexibility. Next we add the vertical dimension of flexibility.

A *reasoning structure*, rs, is a directed labelled graph. The nodes of rs are partitioned into two sets: sequent nodes and link nodes. The edges of rs go from

[11] In fact, it is often the case that some bridge rules contain some "new" constraints, i.e. constraints not present in any component reasoning theory. So, technically speaking, our definition presents some inadequacy to fully capture the process of composing reasoning theories. The refinement of our definition is part of future theoretical work.

link nodes to sequent nodes or from sequent nodes to link nodes. Each link node has a unique incoming node (the *conclusion* or *goal*) and the outgoing nodes (the *premisses* or *subgoals*) are ordered as a sequence. Sequent nodes are labelled by sequents and link nodes are labelled by *justifications*. One kind of justification is a rule application, represented by a rule identifier and a set of constraints. We call link nodes with such justifications, *rule application links*. Another kind of justification is a 4-tuple consisting of a set of constraints, an instantiation map, a sequence of sequent nodes, and a reasoning structure. The instantiation map relates schematic parts of the nested structure to those of its containing structure, as we will see below. The sequent nodes are the nodes in the nested reasoning structure which correspond to the premiss and conclusion nodes of the labelled link node. We call link nodes with such justifications, *nesting links*.

Formally, the set, $\mathbf{Rs}_0[Rth, SN, LN]$, of *basic* (i.e. those with no nesting links) reasoning structures is the set of structures

$$rs = \langle Sn, Ln, g, sg, sL, lL\rangle$$

such that:

(1) $Sn \in \mathrm{P}_\omega(SN)$ is the set of sequent nodes of rs, and $Ln \in \mathrm{P}_\omega(LN)$ is the set of link nodes of rs;

(2) $g : [Ln \to Sn]$ maps each link node to its associated goal sequent node;

(3) $sg : [Ln \to Sn^*]$ maps each link node to its (possibly empty) associated sequence of subgoal sequent nodes;

(4) $sL : [Sn \to S]$ is the sequent node labelling map; if $[sn_1, \ldots, sn_n] \in Sn^*$, we write $sL([sn_1, \ldots, sn_n])$ for $[sL(sn_1), \ldots, sL(sn_n)]$;

(5) $lL : [Ln \to [Id \times \mathrm{P}_\omega(C)]]$ is the link node labelling map. This map must be such that for $ln \in Ln$ if $lL(ln) = \langle id, \widetilde{c}\rangle$, $\overline{s} = sL(sg(ln))$, and $s = sL(g(ln))$, then $\langle \overline{s}, s, \widetilde{c}\,'\rangle \in \widetilde{r}\,(id)$ for some $\widetilde{c}\,'$ such that $\widetilde{c} \models \widetilde{c}\,'$.

The set, $\mathbf{Rs}[Rth, SN, LN]$, of reasoning structures is defined by allowing successively deeper levels of nesting starting with basic reasoning structures at level 0, as follows:

$$\mathbf{Rs}[Rth, SN, LN] = \bigcup_{n \in \mathbf{Nat}} \mathbf{Rs}_n[Rth, SN, LN]$$

where $\mathbf{Rs}_n[Rth, SN, LN]$ is the set of reasoning structures of level n. The reasoning structures of level 0 are the basic reasoning structures defined above. The reasoning structures of level $n+1$ are the structures

$$rs = \langle Sn, Ln, g, sg, sL, lL\rangle$$

such that conditions (1)-(4) above hold, and

(5_{n+1}) $lL : [Ln \to [Id \times \mathrm{P}_\omega(C)] + [\mathrm{P}_\omega(C) \times I \times (Sn^* \times Sn) \times \mathbf{Rs}_n[Rth, SN, LN]]]$ such that if ln is a rule application link then condition (5) for basic reasoning structures holds and if ln is a nesting link with $lL(ln) = \langle \widetilde{c}\,, \iota, \langle \overline{sn}\,, sn\rangle, rs'\rangle$ and $rs' = \langle Sn', Ln', g', sg', sL', lL'\rangle$ then $\overline{sn} \diamond [sn] \in (Sn')^*$, and $sL'(\overline{sn} \diamond [sn])\,[\iota] = sL(sg(ln)) \diamond [sL(g(ln))]$.

Instantiations are applied to reasoning structures in the following way. If $rs = \langle Sn, Ln, g, sg, sL, lL\rangle \in \mathbf{Rs}[Rth, SN, LN]$ and $\iota \in I$, then

$$rs\,[\iota] = \langle Sn, Ln, g, sg, sL\,[\iota]\,, lL'\rangle$$

where, for $ln \in Ln$

(ra) if $lL(ln) = \langle id, \widetilde{c}\rangle$, then $lL'(ln) = \langle id, \widetilde{c}\,[\iota]\,\rangle$,

(nest) if $lL(ln) = \langle \widetilde{c}\,, \iota_1, \langle \overline{sn}\,, sn\rangle, rs_1\rangle$, then $lL'(ln) = \langle \widetilde{c}\,[\iota]\,, \iota \circ \iota_1, \langle \overline{sn}\,, sn\rangle, rs_1\rangle$.

A reasoning structure is a *derivation* of a conclusion sequent from a set of assumption sequents, if it represents a traditional proof figure. That is, if it satisfies conditions (1)-(5) below.

(1) Each rule application link has no unsolved constraints.

(2) Each sequent node is the conclusion of at most one inference (link node).

(3) There is a unique sequent node that does not occur as the premiss of any inference. The sequent labelling this node is the *conclusion* of the derivation. The sequents labelling occurrences which are not the conclusion of any inference are the *open assumptions.*

(4) The underlying graph is acyclic.

(5) For each nesting link, the associated tuple $\langle \widetilde{c}\,, \iota, \langle \overline{sn}\,, sn\rangle, rs\rangle$ is such that $\widetilde{c}$ is the empty set and the reasoning structure $rs\,[\iota]$ is a derivation with conclusion node sn and open assumption nodes $\overline{sn}$.

A sequent s is *Rth-derivable* from a set of sequents $\widetilde{s}$ if there exists a derivation $rs \in \mathbf{Rs}[Rth, SN, LN]$ with conclusion s and open assumptions contained in $\widetilde{s}$. A reasoning structure is a *proof* if it is a derivation with no open assumptions. A sequent s is *Rth-provable* if it is *Rth*-derivable from the empty set of sequents.

It is possible to prove two theorems showing that, for consideration only of derivability, nesting links and sharing of subderivations can be eliminated (vertical and horizontal unfolding):

(1) if $rs \in \mathbf{Rs}[Rth, SN, LN]$ is a derivation of s from $\widetilde{s}$ then we can find a level 0 derivation $rs_0 \in \mathbf{Rs}_0[Rth, SN, LN]$ of s from $\widetilde{s}$;

(2) if rs is a level 0 derivation of s from $\widetilde{s}$ then there exists a level 0 derivation rs' of s from $\widetilde{s}$ such that the directed graph underlying rs' is a tree.

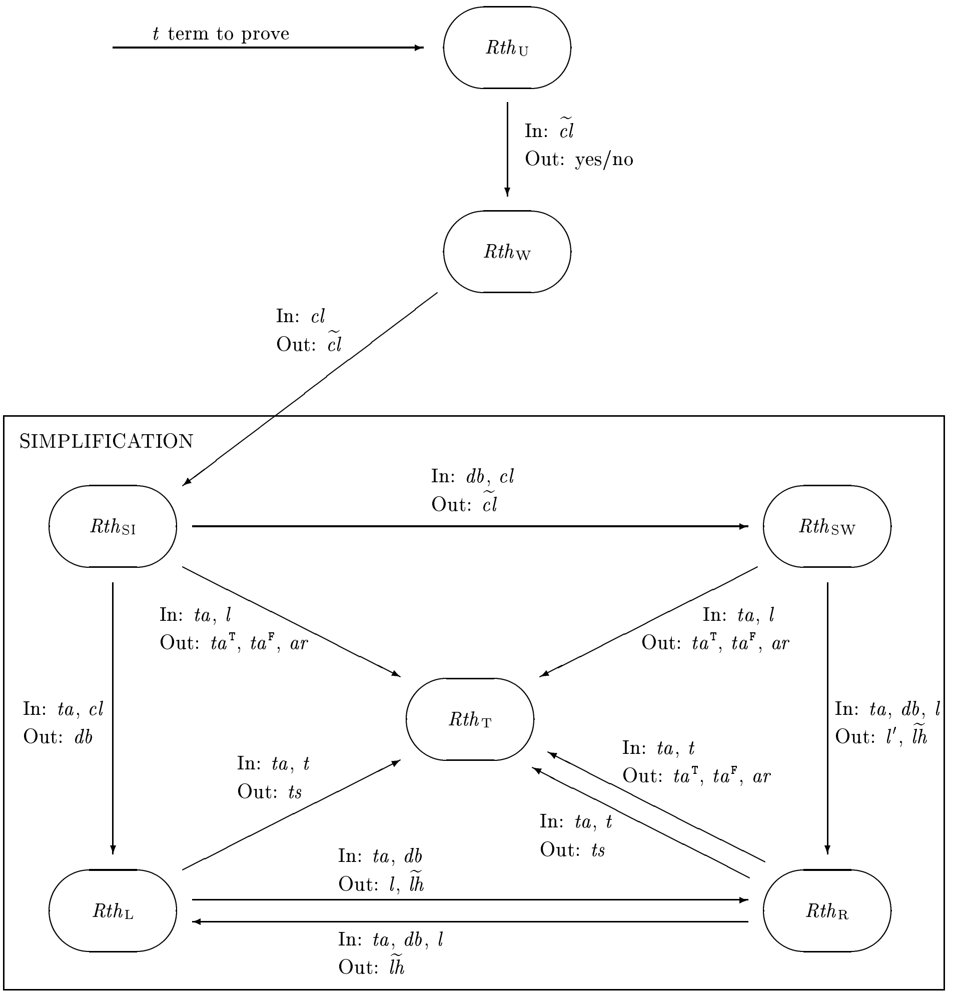

Figure 3. The Reasoning Theory Rth_{NQTHM}

4. The Reasoning Theory Rth_{NQTHM}

In order to make the module structure explicit, we present the NQTHM reasoning theory

$$Rth_{\mathrm{NQTHM}} = \langle Ssys_{\mathrm{NQTHM}}, Id_{\mathrm{NQTHM}}, \widetilde{r}_{\mathrm{NQTHM}} \rangle$$

as a composite reasoning theory constructed out of some component reasoning theories

$$Rth_{\mathrm{NQTHM}} = glueRth([Rth_{\mathrm{U}}, Rth_{\mathrm{W}}, Rth_{\mathrm{SI}}, Rth_{\mathrm{SW}}, Rth_{\mathrm{T}}, Rth_{\mathrm{L}}, Rth_{\mathrm{R}}, \ldots], Id_B, \widetilde{r}_B)$$

where each component reasoning theory corresponds to a reasoning module of NQTHM, and $\widetilde{r}_B$ is a set of bridge rules. The ellipses in the above definition are placeholders for other reasoning theories which we do not specify, as we will explain below.

Figure 3 provides an overview of the organization of Rth_{NQTHM}. Rth_{U} corresponds to the top level module of NQTHM, i.e. the module in charge of interaction with the user when conjectures to be proved are submitted to the system. Rth_{W} corresponds to the master module of the waterfall, i.e. the module that maintains the top and the pool, and controls the invocations of the inference processes. Rth_{SI} corresponds to the high-level controller of the simplifier. Finally Rth_{SW}, Rth_{T}, Rth_{L}, and Rth_{R} respectively correspond to the sweeper, the typeset specialist, the linear arithmetic specialist, and the rewriter. Each arrow in Figure 3 between component reasoning theories corresponds to an invocation of one NQTHM module by the other. These invocations are formalized in our framework using appropriate bridge rules between the corresponding reasoning theories. Arrows of this kind are labelled by our representation of the logical information passed (In) and returned (Out) in the invocation (see the bridge rules below).

The reasoning theory level description of NQTHM can be done in many ways, and at many levels of detail. The description given here is aimed at understanding the integration of the modules of the simplifier. We do not consider therefore the other inference processes: we do not specify the reasoning theories corresponding to modules of these other inference processes, and furthermore we do not specify those bridge rules of $\widetilde{r}_B$ which relate sequents belonging to such reasoning theories.

The sequent system and rules of Rth_{NQTHM} are described in detail in the next two subsections. The data structures manipulated by NQTHM contain additional (non-logical) information used to control the heuristic proof strategies. We have omitted this information for the present, as we are not treating issues of control in this paper.

4.1. The Sequent System $Ssys_{\mathrm{NQTHM}}$

The Rth_{NQTHM} sequent system

$$Ssys_{\mathrm{NQTHM}} = \langle S_{\mathrm{NQTHM}}, C_{\mathrm{NQTHM}}, \models_{\mathrm{NQTHM}}, I_{\mathrm{NQTHM}}, \_\,\texttt{[\_]}_{\mathrm{NQTHM}} \rangle$$

is defined as

$$Ssys_{\mathrm{NQTHM}} = \bigcup_{i \in \{\mathrm{U,W,SI,SW,T,L,R,}\ldots\}} Ssys_i.$$

The sequents and constraints of each component sequent system are terms[12] of some sort of a given algebraic specification. We assume these terms are built with respect

[12] Of course, here by "term" we do not mean those of the logic of NQTHM. However, no ambiguity can arise because, apart from this paragraph, in the rest of the paper we use "term" exclusively to refer to NQTHM terms.

to a given sorted set of variables, so that instantiations are just finite mappings from variables to terms (of corresponding sorts), and instantiation composition, instantiation application, and the identity instantiation, are defined as obvious.

4.1.1. Sequents

We structure the sequents of a component NQTHM sequent system as pairs consisting of (our representation of) an NQTHM history $h \in H$, as in NQTHM all reasoning takes place in the context of some NQTHM history, and a current conjecture. The history and the conjecture are separated by a turnstyle which is labelled to identify the component sequent system.

U-sequents, i.e. sequents of the sequent system $Ssys_{\mathrm{U}}$, have the form

$$h \vdash_{\mathrm{U}} t$$

where the conjecture is a term $t \in Trm$. A U-sequent asserts that the formula represented by t is a consequence of the theory defined by h.

W-sequents have the form

$$h \vdash_{\mathrm{W}} \widetilde{cl}$$

where the conjecture $\widetilde{cl} \in \mathrm{P}_\omega(Cl)$ is a set of clauses (considered conjunctively). The interpretation of W-sequents is analogous to U-sequents.

SI-sequents have the form

$$h \vdash_{\mathrm{SI}} cl \leftarrow \widetilde{cl}$$

where in the conjecture cl and $\widetilde{cl}$ roughly correspond to the input and output of the simplifier, respectively. An SI-sequent asserts that cl is a consequence of h if $\widetilde{cl}$ is such.

SW-sequents have the form

$$h \vdash_{\mathrm{SW}} db;\ \widetilde{cl} \leftarrow \widetilde{cl}'$$

where the conjecture is decomposed into local context information and focus expression. The local context is a polynomial database $db \in Db$ (see below), and the focus expression is $\widetilde{cl} \leftarrow \widetilde{cl}'$ where $\widetilde{cl}$ represents the clauses whose literals are being swept and $\widetilde{cl}'$ represents the output of the sweeper. An SW-sequent asserts that $\widetilde{cl}$ is a consequence of h if $\widetilde{cl}'$ is such under the assumptions in db.

T-sequents have two possible forms

$$h \vdash_{\mathrm{T}}^{\mathrm{TS}} ta;\ t \to ts,$$

$$h \vdash_{\mathrm{T}}^{\mathrm{ASM}} ta;\ t \to ta^{\mathtt{T}}, ta^{\mathtt{F}}, ar,$$

where in the conjecture, the local context is a type alist $ta \in Ta$ and the focus expression is: in the first case $t \to ts$ where $t \in Trm$ is a term, and $ts \in Ts$ is a typeset; in the second case $t \to ta^{\mathtt{T}}, ta^{\mathtt{F}}, ar$ where $t \in \mathit{Trm}$ is a term, $ta^{\mathtt{T}}, ta^{\mathtt{F}} \in \mathit{Ta}$ are two type alists, and $ar \in Ar$ is an assumption response, where $Ar = \{\mathtt{T}, \mathtt{F}, \mathtt{ANY}\}$. A T-sequent of the first form asserts that in h, under the assumptions in ta, t has typeset ts. A T-sequent of the second form asserts that in h, under the assumptions in ta, t must be true or must be false or may be any of them if (respectively) ar is $\mathtt{T}$ or $\mathtt{F}$ or $\mathtt{ANY}$, and in the last case $ta^{\mathtt{T}}$ and $ta^{\mathtt{F}}$ are the type alists obtained by adding to ta the assumption that t is (respectively) true or false.

L-sequents have the form

$$h \vdash_{\mathrm{L}} ta;\ pi \to pi'$$

where the local context is a type alist ta, and the focus is $pi \to pi'$ where $pi, pi' \in Pi$ are two polynomial information structures (defined below). An L-sequent asserts that (the linear inequalities represented by) pi' follows from pi in the theory h under the typeset assumptions in ta.

R-sequents have the form

$$h \vdash_{\mathrm{R}} ta, db;\ t =_m t'\ ; \widetilde{lh}$$

where the local context consists of ta and db (representing the typeset assumptions and polynomial database of the current rewriting context), and the focus expression consists of two terms $t, t' \in \mathit{Trm}$, a modality of equivalence $m \in M$ (where $M = \{\mathtt{I}, \mathtt{B}\}$), and a set of terms $\widetilde{lh} \in \mathrm{P}_\omega(\mathit{Trm})$ (which constitutes the linearization hypotheses returned by the rewriter). An R-sequent asserts that in h, under the assumptions in ta, db and $\widetilde{lh}$, t is identically equivalent (if m is $\mathtt{I}$) or propositionally (if m is $\mathtt{B}$) equivalent to t'.

As it can be noticed from the above description, the sequents share some component objects. For instance the conjectures of SI and W both contain objects of $\mathrm{P}_\omega(\mathit{Cl})$. The next table summarizes this sharing showing the objects composing each sequent.

Sequents	Components
U	H, Trm
W	$H, \mathrm{P}_\omega(\mathit{Cl})$
SI	$H, \mathit{Cl}, \mathrm{P}_\omega(\mathit{Cl})$
SW	$H, \mathit{Db}, \mathrm{P}_\omega(\mathit{Cl})$
T	$H, \mathit{Ta}, \mathit{Trm}, \mathit{Ts}, Ar$
L	H, Ta, Pi
R	$H, \mathit{Ta}, \mathit{Db}, \mathit{Trm}, M, \mathrm{P}_\omega(\mathit{Trm})$

The sharing actually happens also at a lower level of detail. That is, the above components share further sub-components, as we will explain in the sequel.

A clause is a finite set of terms, so in fact $\mathit{Cl} = \mathrm{P}_\omega(\mathit{Trm})$.

A polynomial $p \in \mathit{Poly}$ is an object

$$\langle i, i_1 * t_1 + \cdots + i_n * t_n, \{lh_1, \ldots, lh_m\}\rangle$$

where $n \geq 0$, $i, i_1, \ldots, i_n$ are integers (the constant and the coefficients), $t_1, \ldots, t_n$ are terms (the multiplicands), and $lh_1, \ldots, lh_m$ are terms (the linearization hypotheses).

A database is a finite set of polynomials, so in fact $Db = \mathrm{P}_\omega(\mathit{Poly})$.

A polynomial information structure $pi \in Pi$ is an object

$$\langle db, \widetilde{l}, \widetilde{\widetilde{p}} \rangle$$

where db is a database, $\widetilde{l}$ is a finite set of literals (terms), and $\widetilde{\widetilde{p}}$ is a finite set of finite sets of polynomials. Roughly speaking, the polynomial information structures manipulated by the linear arithmetic specialist consist of the current database, the literals to be linearized, and the polynomials obtained by linearization, respectively.

A type alist $ta \in Ta$ is a finite set of pairs $\langle t, ts\rangle$, where t is a term and $ts \in Ts$ is a typeset, such that no two distinct pairs in ta have the same term as first component.

A history $h \in H$ is a finite sequence of objects representing events. We do not go into the details of these objects. The only important thing is that a history includes function definitions, rewrite rules, and linear rules.

A function definition $def \in Def$ is an object

$$\langle f, [v_1, \ldots, v_n], body\rangle$$

where f is a function symbols, $v_1, \ldots, v_n$ $(n \geq 0)$ are distinct variables, and $body$ is a term satisfying certain properties.

A rewrite rule $rr \in Rr$ is an object

$$\langle [l_1, \ldots, l_n], m, lhs, rhs\rangle$$

where $l_1, \ldots, l_n$ $(n \geq 0)$ are terms (the hypotheses), m is a modality of equivalence, and lhs and rhs are terms (the left- and right-hand side of the conclusion).

A linear rule $lr \in Lr$ is an object

$$\langle [l_1, \ldots, l_n], t\rangle$$

where $l_1, \ldots, l_n$ $(n \geq 0)$ are terms (the hypotheses), and t is a term of the form (`LESSP` *lhs rhs*) or (`NOT` (`LESSP` *lhs rhs*)).

Finally, we have to define another object that is used in the sequel. A substitution $sb \in Sb$ is a finite set of pairs $\langle v, t\rangle$, where v is a variable and t is a term, such that no two distinct pairs in sb have the same variable as first component.

4.1.2. Constraints

Analogously to sequents, we do not explicitly present the algebraic specification for constraints, but instead we present constraints as predicates (i.e. relations) over certain sets. To ease readability, we present constraints along with the rules which use them, in the following subsections.

4.2. The Rth_{NQTHM} Rules

This subsection is devoted to the rules of Rth_{NQTHM}. We start by describing the internal rules of the Rth_{NQTHM} component reasoning theories. We conclude introducing the Rth_{NQTHM} bridge rules. We describe the functions and predicates we use when needed. To provide some intuition for each rule we explain the NQTHM reasoning step the rule is intended to describe. Notice that we present these rules using the classical forward form, i.e. from the premisses to the conclusion, even though the rules are adirectional and their intended use in the prover is backward.

Each rule is presented using the following notation:

$$\frac{\begin{array}{c} s_1 \\ \vdots \\ s_n \end{array}}{s} \quad \langle \mathtt{id}, \widetilde{c} \rangle$$

where $s, s_1, \ldots, s_n$ are sequents, $\widetilde{c}$ is a set of constraints and $\mathtt{id}$ is a rule identifier. Schemas like the one above should be thought of as presenting the rule generators for the rules considered. For instance the schema above presents the rule generator

$$\{\langle [s_1, \ldots, s_n], s, \widetilde{c} \rangle\}$$

of the rule whose identifier is `id`.

4.2.1. The Rth_{U} Internal Rules

The reasoning theory for the top level module of NQTHM only possesses one internal rule, `expabr`, which formalizes the expansion of the abbreviations in the conjecture supplied by the user.

$$\frac{h \vdash_{\mathrm{U}} expabr(h, t)}{h \vdash_{\mathrm{U}} t} \quad \langle \mathtt{expabr}, \emptyset \rangle$$

The function

$$expabr : H \times Trm \to Trm$$

returns the term obtained by expanding the abbreviations in the input term, under the input theory.

4.2.2. The Rth_{W} Internal Rules

Most of the work performed by the waterfall process corresponds to its invocations of the inference processes, and is therefore modeled using bridge rules. What remains consists of two simple “internal” tasks that are described by the two following rules.

The rule `elimsbs` formalizes the elimination of a subsumed clause in the pool.

$$\frac{h \vdash_{\mathrm{W}} \widetilde{cl} \cup \{cl\}}{h \vdash_{\mathrm{W}} \widetilde{cl} \cup \{cl\} \cup \{cl'\}} \quad \langle \texttt{elimsbs}, \{\mathit{Subsumes}(cl,cl')\}\rangle$$

In this rule the predicate

$$\mathit{Subsumes} \subseteq \mathit{Cl} \times \mathit{Cl}$$

consists of all the pairs $\langle cl, cl' \rangle$ of clauses such that cl' is subsumed by cl.

The rule `qed` expresses the fact that when there are no clauses (subgoals) left both in the pool and in the top of the waterfall, the initial conjecture has been proved.

$$\frac{}{h \vdash_{\mathrm{W}} \emptyset} \quad \langle \texttt{qed}, \emptyset \rangle$$

4.2.3. The Rth_{SI} Internal Rules

There are three internal rules in Rth_{SI} which formalize the elimination of trivial equations from the clause given to the simplifier as input.

$$\frac{h \vdash_{\mathrm{SI}} \mathit{sbt}(cl, \{\langle v, t \rangle\}) \leftarrow \widetilde{cl}}{h \vdash_{\mathrm{SI}} cl \cup \{(\texttt{NOT (EQUAL } v\ t\texttt{))}\} \leftarrow \widetilde{cl}} \quad \langle \texttt{remNEvt}, \{\mathit{Var}(v), \mathit{NotOcc}(v,t)\}\rangle$$

$$\frac{h \vdash_{\mathrm{SI}} \mathit{sbt}(cl, \{\langle v, t \rangle\}) \leftarrow \widetilde{cl}}{h \vdash_{\mathrm{SI}} cl \cup \{(\texttt{NOT (EQUAL } t\ v\texttt{))}\} \leftarrow \widetilde{cl}} \quad \langle \texttt{remNEtv}, \{\mathit{Var}(v), \mathit{NotOcc}(v,t)\}\rangle$$

$$\frac{h \vdash_{\mathrm{SI}} \mathit{sbt}(cl, \{\langle v, \texttt{F} \rangle\}) \leftarrow \widetilde{cl}}{h \vdash_{\mathrm{SI}} cl \cup \{v\} \leftarrow \widetilde{cl}} \quad \langle \texttt{remv}, \{\mathit{Var}(v)\}\rangle$$

In these rules the predicate

$$\mathit{Var} \subseteq \mathit{Trm}$$

consists of all the variables, and the predicate

$$\mathit{NotOcc} \subseteq \mathit{Trm} \times \mathit{Trm}$$

consists of all the pairs $\langle t_1, t_2 \rangle$ such that t_1 does not occur in t_2. The function

$$\mathit{sbt} : \mathit{Trm} \times \mathit{Sb} \rightarrow \mathit{Trm}$$

applies a substitution to a term. By abuse of notation we assume that the function *sbt* is defined (by homomorphic lifting) also on structures built from terms, e.g. clauses.

4.2.4. The Rth_{SW} Internal Rules

The unique internal rule we introduce below expresses the trivial fact that a set of clauses is derivable from itself. Its intended use is when all the literals of $\widetilde{cl}$ have been swept and so the sweeper stops the reasoning process to "return" a value.

$$\frac{}{h \vdash_{\mathrm{SW}} db;\ \widetilde{cl} \leftarrow \widetilde{cl}} \quad \langle\texttt{reflsw},\emptyset\rangle$$

4.2.5. The Rth_{T} Internal Rules

In our formalization the logical services provided by the typeset specialist are modeled by the two rules presented below.

$$\frac{}{h \vdash_{\mathrm{T}}^{\mathrm{TS}} ta;\ t \to tscomp(h, ta, t)} \quad \langle\texttt{tscomp},\emptyset\rangle$$

The function

$$tscomp : H \times Ta \times Trm \to Ts$$

computes the typeset associated to the input term under the input theory and the assumptions in the input type alist.

$$\frac{}{h \vdash_{\mathrm{T}}^{\mathrm{ASM}} ta;\ t \to asmT(h, ta, t), asmF(h, ta, t), aresp(h, ta, t)} \quad \langle\texttt{assumeTF},\emptyset\rangle$$

The functions

$$asmT : H \times Ta \times Trm \to Ta,$$

$$asmF : H \times Ta \times Trm \to Ta,$$

$$aresp : H \times Ta \times Trm \to Ar,$$

respectively return the type alist obtained by adding to the input one the assumption that the input term is true, the type alist obtained by adding to the input one the assumption that the input term is false, and the assumption response `T` (if the input term must be true) or `F` (if it must be false) or `ANY` (if it may be either true or false); in case the term must be true or must be false, *asmT* and *asmF* just return the input type alist.

4.2.6. The Rth_{L} Internal Rules

For the sake of readability we have organized these rules in subsections each corresponding to a certain activity performed by the linear arithmetic specialist. In the first subsection there are rules corresponding to general logical properties of the data structures manipulated by this module. Then we describe the rules for manipulating a polynomial database by linear arithmetic reasoning. Finally we introduce the rules that are used for pushing literals into the polynomial database, i.e. for adding to the polynomial database polynomials obtained from the linearization of literals.

4.2.6.1. Logical Reasoning

The rule `reflla` states that a polynomial information structure can be derived from itself. Its intended use is when the linear arithmetic specialist has finished its reasoning task and returns a value.

$$\frac{}{h \vdash_{\mathrm{L}} ta;\ pi \rightarrow pi} \quad \langle\texttt{reflla},\emptyset\rangle$$

The rule `transla` states, informally speaking, that the linear arithmetic reasoning performed by the linear arithmetic module is transitive.

$$\frac{\begin{array}{l} h \vdash_{\mathrm{L}} ta;\ pi_1 \rightarrow pi_2 \\ h \vdash_{\mathrm{L}} ta;\ pi_2 \rightarrow pi_3 \end{array}}{h \vdash_{\mathrm{L}} ta;\ pi_1 \rightarrow pi_3} \quad \langle\texttt{transla},\emptyset\rangle$$

4.2.6.2. Linear Arithmetic Reasoning

The rule `crossmult` adds to a polynomial database the polynomial obtained by cross-multiplying and adding two polynomials.

$$\frac{}{h \vdash_{\mathrm{L}} ta;\ \langle db, \widetilde{l}, \widetilde{\widetilde{p}}\ \rangle \rightarrow \langle db \cup \{(i_1 \otimes p_1 \oplus i_2 \otimes p_2)\}, \widetilde{l}, \widetilde{\widetilde{p}}\ \rangle} \quad \langle\texttt{crossmult},\widetilde{c}\rangle$$

where $\widetilde{c}$ is the set of constraints

$$\begin{array}{l} p_1 \in db \\ p_2 \in db. \end{array}$$

The function

$$\otimes : Int \times Poly \rightarrow Poly$$

returns the polynomial obtained multiplying the constant and all the coefficients of the input polynomial by the input integer (and, of course, leaving the linearization hypotheses unaltered). The function

$$\oplus : Poly \times Poly \rightarrow Poly$$

returns the polynomial obtained from the two input polynomials by adding their constants, “merging” their multiplicands and coefficients in the obvious way, and taking the union of their sets of linerization hypotheses

The rule addnna is used to add the non-negative-assumption for a term to a database.

$$\frac{}{h \vdash_{\mathrm{L}} ta;\ \langle db, \widetilde{l}, \widetilde{\widetilde{p}}\ \rangle \rightarrow \langle db \cup \{\langle 0, -1 * t, \emptyset \rangle\}, \widetilde{l}, \widetilde{\widetilde{p}}\ \rangle} \quad \langle \texttt{addnna}, \emptyset \rangle$$

The rule elimvacuous is used to get rid of vacuous polynomials.

$$\frac{}{h \vdash_{\mathrm{L}} ta;\ \langle db \cup \{p\}, \widetilde{l}, \widetilde{\widetilde{p}}\ \rangle \rightarrow \langle db, \widetilde{l}, \widetilde{\widetilde{p}}\ \rangle} \quad \langle \texttt{elimvacuous}, \{\mathit{Vacuous}(p)\} \rangle$$

The predicate

$$\mathit{Vacuous} \subseteq \mathit{Poly}$$

consists of all the polynomials $\langle i, i_1 * t_1 + \cdots + i_n * t_n, \widetilde{lh}\ \rangle$ such that $i \leq 0$ and $i_j \leq 0$ for each $1 \leq j \leq n$.

4.2.6.3. Pushing Literals into the Polynomial Database

The activity of pushing a literal into a polynomial database can be divided in two parts. First the literal is linearized. Then the resulting polynomials are inserted into the database according to certain criteria, as explained in §2.

The rule linearize expresses the first part of this process (notice that in this rule the result is put in the third component of the polynomial information structure).

$$\frac{}{h \vdash_{\mathrm{L}} ta;\ \langle db, \widetilde{l} \cup \{l\}, \emptyset \rangle \rightarrow \langle db, \widetilde{l}, \mathit{lin}(h, ta, l) \rangle} \quad \langle \texttt{linearize}, \emptyset \rangle$$

The function

$$\mathit{lin} : H \times \mathit{Ta} \times \mathit{Trm} \rightarrow \mathrm{P}_{\omega}(\mathrm{P}_{\omega}(\mathit{Poly}))$$

returns the (positive) linearization of an NQTHM term (under the input theory and type alist), which consists in a set of sets of polynomials. Notice that the linearization procedure makes use of some typeset reasoning (cf. p. 44 [3]), in fact *lin* also receives the current type alist as input; this invocation of the typeset specialist by the linear arithmetic specialist is thus hidden in *lin* and not formalized as a bridge rule.

The rules shown below formalize the process of adding the result of the linearization of a literal to the database. The rule pushconj treats the case in which the result of the linearization is a singleton set consisting of a set of polynomials. In this case the polynomials are simply added to the database.

$$\frac{}{h \vdash_{\mathrm{L}} ta;\ \langle db, \widetilde{l}, \{\widetilde{p}\,\}\rangle \rightarrow \langle db \cup \widetilde{p}\,, \widetilde{l}, \emptyset\rangle}\ \langle\texttt{pushconj},\emptyset\rangle$$

As already explained in §2, in case the result of the linearization is a pair of alternatives (i.e. it consists in a doubleton set containing two sets of polynomials), the linear arithmetic specialist employs the following strategy. If pushing the second alternative into the database and augmenting with linear rules produces an impossible polynomial, then the first alternative is pushed into the database (and the second one is discarded), after having added the linearization hypotheses of the impossible polynomial found to those of each polynomial of the first alternative; this is formalized by the rule `pushdisj1`. Otherwise, if pushing the first alternative into the database and augmenting with linear rules produces an impossible polynomial, then the second alternative is pushed into the database (and the first one is discarded), after having added the linearization hypotheses of the impossible polynomial found to those of each polynomial of the second alternative; this is formalized by the rule `pushdisj2`. Otherwise, the alternatives are simply discarded, as formalized by the rule `elimdisj`.

$$\frac{h \vdash_{\mathrm{L}} ta;\ \langle db \cup \widetilde{p}_{\,2}, \emptyset, \emptyset\rangle \rightarrow \langle db' \cup \{p\}, \emptyset, \emptyset\rangle}{h \vdash_{\mathrm{L}} ta;\ \langle db, \widetilde{l}, \{\widetilde{p}_{\,1}, \widetilde{p}_{\,2}\}\rangle \rightarrow \langle db \cup \mathit{addlhyps}(\widetilde{p}_{\,1}, \widetilde{lh}\,), \widetilde{l}, \emptyset\rangle}\ \langle\texttt{pushdisj1},\widetilde{c}\,\rangle$$

where $\widetilde{c}$ is the set of constraints

$$\begin{array}{l} \mathit{Impossible}(p) \\ p \sim \langle i, i_1 * t_1 + \cdots + i_n * t_n, \widetilde{lh}\,\rangle. \end{array}$$

The predicate

$$\mathit{Impossible} \subseteq \mathit{Poly}$$

consists of all the polynomials $\langle i, i_1 * t_1 + \cdots + i_n * t_n, \widetilde{lh}\,\rangle$ such that $i > 0$ and $i_j \geq 0$ for each $1 \leq j \leq n$. The function

$$\mathit{addlhyps} : \mathrm{P}_\omega(\mathit{Poly}) \times \mathrm{P}_\omega(\mathit{Trm}) \rightarrow \mathrm{P}_\omega(\mathit{Poly})$$

adds the input set of terms to the linearization hypotheses of each polynomial of the input set of polynomials.

$$\frac{h \vdash_{\mathrm{L}} ta;\ \langle db \cup \widetilde{p}_{\,1}, \emptyset, \emptyset\rangle \rightarrow \langle db' \cup \{p\}, \emptyset, \emptyset\rangle}{h \vdash_{\mathrm{L}} ta;\ \langle db, \widetilde{l}, \{\widetilde{p}_{\,1}, \widetilde{p}_{\,2}\}\rangle \rightarrow \langle db \cup \mathit{addlhyps}(\widetilde{p}_{\,2}, \widetilde{lh}\,), \widetilde{l}, \emptyset\rangle}\ \langle\texttt{pushdisj2},\widetilde{c}\,\rangle$$

where $\widetilde{c}$ is the set of constraints

$$\begin{array}{l} \mathit{Impossible}(p) \\ p \sim \langle i, i_1 * t_1 + \cdots + i_n * t_n, \widetilde{lh} \rangle. \end{array}$$

$$\frac{}{h \vdash_{\mathrm{L}} ta;\ \langle db, \widetilde{l}, \{\widetilde{p}_1, \widetilde{p}_2\}\rangle \to \langle db, \widetilde{l}, \emptyset\rangle} \quad \langle\mathtt{elimdisj},\emptyset\rangle$$

4.2.7. The Rth_{R} Internal Rules

We have organized the internal rules of Rth_{R} in various subsections. In the first subsection we present rules describing some general logical properties of the rewriting activity. Then there are two subsections respectively describing rules for the manipulation of `IF`-expressions and `EQUAL`-expressions. Finally there are two subsections devoted to the two most important activities performed by the rewriter, i.e. the use of lemmas and axioms to rewrite expressions, and the opening up of functions.

4.2.7.1. Logical Reasoning

The rule `reflr` states the law of reflexivity for the reasoning performed by the rewriter.

$$\frac{}{h \vdash_{\mathrm{R}} ta, db;\ t =_m t\ ; \emptyset} \quad \langle\mathtt{reflr},\emptyset\rangle$$

The rule `transr` expresses the law of transitivity for the reasoning performed by the rewriter.

$$\frac{\begin{array}{c} h \vdash_{\mathrm{R}} ta, db;\ t_1 =_m t_2\ ; \widetilde{lh}\,' \\ h \vdash_{\mathrm{R}} ta, db;\ t_2 =_m t_3\ ; \widetilde{lh}\,'' \end{array}}{h \vdash_{\mathrm{R}} ta, db;\ t_1 =_m t_3\ ; \widetilde{lh}\,' \cup \widetilde{lh}\,''} \quad \langle\mathtt{transr},\emptyset\rangle$$

Notice how the information is propagated in the rule `transr`: the linearization hypotheses used in the subgoals are collected in the conclusion (like dependencies in Natural Deduction systems).

The rule `congr` formalizes the law of congruence for the reasoning performed by the rewriter.

$$\frac{h \vdash_{\mathrm{R}} ta, db;\ t_i =_{\mathtt{I}} t_i'\ ; \widetilde{lh}}{h \vdash_{\mathrm{R}} ta, db;\ (f\ t_1\ \ldots\ t_i\ \ldots\ t_n) =_m (f\ t_1\ \ldots\ t_i'\ \ldots\ t_n)\ ; \widetilde{lh}} \quad \langle\mathtt{congr},\emptyset\rangle$$

Notice that the subterm is rewritten using the identity preserving modality.

The meaning of the rule `abstractr` is that identical rewriting is "stronger" then propositional rewriting, i.e. results obtained preserving identical equivalence of terms can be safely assumed when the propositional equivalence of terms has to be maintained.

$$\frac{h \vdash_{\mathrm{R}} ta, db;\ t =_{\mathtt{I}} t'\ ; \widetilde{lh}}{h \vdash_{\mathrm{R}} ta, db;\ t =_{\mathtt{B}} t'\ ; \widetilde{lh}}\ \langle\texttt{abstractr},\emptyset\rangle$$

4.2.7.2. Rewriting IF-Expressions

The rule `ifTest` presented below describes how the test of an `IF`-expression is rewritten. Notice that the modality of rewriting preserves the propositional equivalence of the test.

$$\frac{h \vdash_{\mathrm{R}} ta, db;\ t =_{\mathtt{B}} t'\ ; \widetilde{lh}}{h \vdash_{\mathrm{R}} ta, db;\ (\mathtt{IF}\ t\ t_1\ t_2) =_m (\mathtt{IF}\ t'\ t_1\ t_2)\ ; \widetilde{lh}}\ \langle\texttt{ifTest},\emptyset\rangle$$

The next rules formalize the application of two of the three rewrite rules for `IF` explained in §2 (the third rewrite rule is formalized by means of a bridge rule because it involves a call of the typeset specialist; see below).

$$\frac{}{h \vdash_{\mathrm{R}} ta, db;\ (\mathtt{IF}\ t_1\ t_2\ t_2) =_m t_2\ ; \emptyset}\ \langle\texttt{if1},\emptyset\rangle$$

$$\frac{}{h \vdash_{\mathrm{R}} ta, db;\ (\mathtt{IF}\ t\ t\ \mathtt{F}) =_m t\ ; \emptyset}\ \langle\texttt{if2},\emptyset\rangle$$

4.2.7.3. Rewriting EQUAL-Expressions

The rule `refEQ` states the simple property of reflexivity satisfied by the function `EQUAL`.

$$\frac{}{h \vdash_{\mathrm{R}} ta, db;\ (\mathtt{EQUAL}\ t\ t) =_m \mathtt{T}\ ; \emptyset}\ \langle\texttt{refEQ},\emptyset\rangle$$

The rules `eq1` and `eq2` formalize two of the rewrite rules for `EQUAL` explained in §2 (the third rewrite rule is formalized by means of a bridge rule because it involves a call of the typeset specialist; see below).

$$\frac{}{h \vdash_{\mathrm{R}} ta, db;\ (\mathtt{EQUAL}\ t_1\ (\mathtt{EQUAL}\ t_2\ t_3)) =_m t\ ; \emptyset}\ \langle\texttt{eq1},\emptyset\rangle$$

where t is the term $(\mathtt{IF}\ (\mathtt{EQUAL}\ t_2\ t_3)\ (\mathtt{EQUAL}\ t_1\ \mathtt{T})\ (\mathtt{EQUAL}\ t_1\ \mathtt{F}))$.

$$\frac{}{h \vdash_{\mathrm{R}} ta, db;\ (\mathtt{EQUAL}\ t\ \mathtt{F}) =_m (\mathtt{IF}\ t\ \mathtt{F}\ \mathtt{T})\ ; \emptyset}\ \langle\texttt{eq2},\emptyset\rangle$$

4.2.7.4. Rewriting a Term with Lemmas

The rule `rewR` formalizes the application of a rewrite rule.

$$\frac{[h \vdash_{\mathrm{R}} ta, db;\ sbt(l_i, sb) =_{\mathtt{B}} \mathtt{T}\ ; \widetilde{lh}_i \mid 1 \leq i \leq n]}{h \vdash_{\mathrm{R}} ta, db;\ sbt(lhs, sb) =_m sbt(rhs, sb)\ ; \bigcup_{i=1}^{n} \widetilde{lh}_i}\ \langle \mathtt{rewR}, \widetilde{c}\rangle$$

where $\widetilde{c}$ is the set of constraints

$$\langle [l_i \mid 1 \leq i \leq n], m, lhs, rhs\rangle \in rr{-}rules(h).$$

The function

$$rr{-}rules : H \to \mathrm{P}_\omega(Rr)$$

returns the set of rewrite rules of the input history. Notice that the hypotheses of the rewrite rule are rewritten preserving propositional equivalence.

When a hypothesis has the form (`NOT` l), the rewriter tries to rewrite l to `F` instead, as formalized by the rule `neghyp`.

$$\frac{h \vdash_{\mathrm{R}} ta, db;\ l =_{\mathtt{B}} \mathtt{F}\ ; \widetilde{lh}}{h \vdash_{\mathrm{R}} ta, db;\ (\mathtt{NOT}\ l) =_{\mathtt{B}} \mathtt{T}\ ; \widetilde{lh}}\ \langle \mathtt{neghyp}, \emptyset\rangle$$

4.2.7.5. Unfolding Function Definitions

The rule `applydef` expresses the unfolding of a function definition.

$$\frac{}{h \vdash_{\mathrm{R}} ta, db;\ (f\ t_1 \ldots\ t_n) =_{\mathtt{I}} sbt(body, \{\langle v_1, t_1\rangle, \ldots, \langle v_n, t_n\rangle\})\ ; \emptyset}\ \langle \mathtt{applydef}, \widetilde{c}\rangle$$

where $\widetilde{c}$ is the set of constraints

$$\langle f, [v_1, \ldots, v_n], body\rangle \in defs(h).$$

The function

$$defs : H \to \mathrm{P}_\omega(Def)$$

returns the set of function definitions of the input history.

4.2.8. The Rth_{NQTHM} Bridge Rules

In this subsection we formalize the integration of the NQTHM modules in terms of bridge rules of the Rth_{NQTHM} reasoning theory. The presentation of these bridge rules is organized in various subsections, each corresponding to the calls performed by a module of NQTHM to other modules.

4.2.8.1. The Calls by the Top Level Module

The invocation of the waterfall by the top level module of NQTHM is represented by the rule `clausify`. The arrow from Rth_{U} to Rth_{W} in Figure 3 shows the logical information involved in this invocation. The input for the waterfall is a set of clauses $\widetilde{cl}$ obtained from the clausification of either the result of expanding the abbreviations in the initial conjecture (at the beginning of a tentative proof), or the initial conjecture itself (in case induction is performed upon the initial conjecture). The output from the waterfall is a yes/no answer. Informally speaking we can say that the waterfall process never provides partial proofs for conjectures, but only reports a success or a failure in determining provability.

$$\frac{h \vdash_{\mathrm{W}} \mathit{cnf}(t)}{h \vdash_{\mathrm{U}} t} \quad \langle \texttt{clausify}, \emptyset \rangle$$

The function

$$\mathit{cnf} : \mathit{Trm} \rightarrow \mathrm{P}_{\omega}(\mathit{Cl})$$

provides the translation from the Rth_{U}'s world, that uses terms to express conjectures, to the Rth_{W}'s world, that manipulates sets of clauses: the returned set of clauses constitutes the conjunctive normal form of the input term.

4.2.8.2. The Calls by the Waterfall

The invocation of the simplifier by the waterfall is formalized by the rule `callsimp` (see the arrow from Rth_{W} to Rth_{SI} in Figure 3).

$$\frac{\begin{array}{c} h \vdash_{\mathrm{SI}} cl \leftarrow \widetilde{cl}_1 \\ h \vdash_{\mathrm{W}} \widetilde{cl} \cup \widetilde{cl}_1 \end{array}}{h \vdash_{\mathrm{W}} \widetilde{cl} \cup \{cl\}} \quad \langle \texttt{callsimp}, \emptyset \rangle$$

Notice that informally speaking `callsimp` can be interpreted in a backward way as a substitution of the subgoal cl with the subgoals in $\widetilde{cl}_1$.

4.2.8.3. The Calls by the High-level Controller

The rule `tacontr` expresses the fact that a contradiction has been derived by typeset reasoning, when building the typeset alist obtained from the negation of the literals of clause to be simplified (see the arrow from Rth_{SI} to Rth_{T} in Figure 3). This means that the input clause is true.

$$\frac{[h \vdash_{\mathrm{T}}^{\mathrm{ASM}} ta_{i-1}^{\mathtt{F}};\ l_i \rightarrow ta_i^{\mathtt{T}}, ta_i^{\mathtt{F}}, ar_i \mid 1 \leq i \leq n]}{h \vdash_{\mathrm{SI}} \{l_1, \ldots, l_n\} \leftarrow \emptyset} \quad \langle \texttt{tacontr}, \{\mathtt{T} \in \{ar_1, \ldots, ar_n\}\} \rangle$$

where $ta_0^{\mathtt{F}}$ is $\emptyset$.

The rule `dbcontr` expresses the fact that a contradiction has been derived by linear arithmetic reasoning, when building the polynomial database, after having built the type alist obtained by the negation of the input clause of the simplifier (see the arrow from Rth_{SI} to Rth_{T} and that from Rth_{SI} to Rth_{L}, in Figure 3). This means that the simplifier returns the sets of clauses obtained by splitting the input clause on the linearization hypotheses of the impossible polynomial found.

$$\frac{\begin{array}{c}[h \vdash_{\mathrm{T}}^{\mathrm{ASM}} ta^{\mathtt{F}}_{i-1};\ l_i \to ta^{\mathtt{T}}_i, ta^{\mathtt{F}}_i, ar_i \mid 1 \leq i \leq n] \\ h \vdash_{\mathrm{L}} ta^{\mathtt{F}}_n;\ \langle \emptyset, \{neg(l_1), \ldots, neg(l_n)\}, \emptyset\rangle \to \langle db \cup \{p\}, \emptyset, \emptyset\rangle\end{array}}{h \vdash_{\mathrm{SI}} \{l_1, \ldots, l_n\} \leftarrow split(\{l_1, \ldots, l_n\}, \widetilde{lh}\,)}\ \langle \mathtt{dbcontr}, \widetilde{c}\rangle$$

where $ta^{\mathtt{F}}_0$ is $\emptyset$, and $\widetilde{c}$ is the set of constraints

$$\begin{array}{l} Impossible(p) \\ p \sim \langle i, i_1 * t_1 + \cdots + i_m * t_m, \widetilde{lh}\,\rangle. \end{array}$$

The function

$$neg : Trm \to Trm$$

returns t is the input term has the form (`NOT` t), returns (`NOT` t) (where t is the input term) otherwise. The function

$$split : Cl \times \mathrm{P}_{\omega}(Trm) \to \mathrm{P}_{\omega}(Cl)$$

splits a clause on a set of literals, i.e.

$$split(cl, \widetilde{l}\,) = \bigcup_{l \in \widetilde{l}} \{cl \cup \{l\}\}.$$

The rule `findEq` corresponds to the introduction of an equality derived from the polynomial database into the input clause of the simplifier, after building the type alist and the polynomial database obtained by the negation of the clause (see the arrows from Rth_{SI} to Rth_{T} and from Rth_{SI} to Rth_{L} in Figure 3).

$$\frac{\begin{array}{c}[h \vdash_{\mathrm{T}}^{\mathrm{ASM}} ta^{\mathtt{F}}_{i-1};\ l_i \to ta^{\mathtt{T}}_i, ta^{\mathtt{F}}_i, ar_i \mid 1 \leq i \leq n] \\ h \vdash_{\mathrm{L}} ta^{\mathtt{F}}_n;\ \langle \emptyset, \{neg(l_1), \ldots, neg(l_n)\}, \emptyset\rangle \to \langle db \cup \{p_1, p_2\}, \emptyset, \emptyset\rangle\end{array}}{h \vdash_{\mathrm{SI}} \{l_1, \ldots, l_n\} \leftarrow addhyps(\{l_1, \ldots, l_n\}, \widetilde{lh}\,) \cup split(\{l_1, \ldots, l_n\}, \widetilde{lh}\,')}\ \langle \mathtt{findEq}, \widetilde{c}\rangle$$

where $ta^{\mathtt{F}}_0$ is $\emptyset$, and $\widetilde{c}$ is the set of constraints

$$\begin{array}{l} Mates(p_1, p_2, t, t') \\ p_1 \sim \langle i^1, i^1_1 * t^1_1 + \cdots + i^1_{m_1} * t^1_{m_1}, \widetilde{lh}_1\rangle \\ p_2 \sim \langle i^2, i^2_1 * t^2_1 + \cdots + i^2_{m_2} * t^2_{m_2}, \widetilde{lh}_2\rangle \\ \widetilde{lh} \sim \widetilde{lh}_1 \cup \widetilde{lh}_2 \cup \{(\mathtt{NUMBERP}\ t)\} \cup \{(\mathtt{NUMBERP}\ t')\} \cup \{(\mathtt{EQUAL}\ t\ t')\} \\ \widetilde{lh}\,' \sim \widetilde{lh}_1 \cup \widetilde{lh}_2. \end{array}$$

The function

$$\mathit{addhyps} : \mathit{Cl} \times \mathrm{P}_\omega(\mathit{Trm}) \to \mathit{Cl}$$

returns the union of the input clause with the negations of all the literals in the input set. The predicate

$$\mathit{Mates} \subseteq \mathit{Poly} \times \mathit{Poly} \times \mathit{Trm} \times \mathit{Trm}$$

consists of all the 4-tuples $\langle p_1, p_2, t, t' \rangle$ such that p_1 and p_2 are mates on t and t' is the conglomerated term corresponding to t, as defined at p. 53–54 of [3].

The rule `swInit` represents the invocation of the sweeper by the high-level controller of the simplifier, after having built the polynomial database obtained by the negation of the input clause. The arrow from $\mathit{Rth}_{\mathrm{SI}}$ to $\mathit{Rth}_{\mathrm{SW}}$ in Figure 3 shows the input-output behavior of the invocation. The input is the input clause cl and the polynomial database db. The output is a set of clauses $\widetilde{cl}$.

$$\frac{\begin{array}{c} [h \vdash_{\mathrm{T}}^{\mathrm{ASM}} ta_{i-1}^{\mathtt{F}};\ l_i \to ta_i^{\mathtt{T}}, ta_i^{\mathtt{F}}, ar_i \mid 1 \leq i \leq n] \\ h \vdash_{\mathrm{L}} ta_n^{\mathtt{F}};\ \langle \emptyset, \{neg(l_1), \ldots, neg(l_n)\}, \emptyset \rangle \to \langle db, \emptyset, \emptyset \rangle \\ h \vdash_{\mathrm{SW}} db;\ \{\{l_1, \ldots, l_n\}\} \leftarrow \widetilde{cl} \end{array}}{h \vdash_{\mathrm{SI}} \{l_1, \ldots, l_n\} \leftarrow \widetilde{cl}} \quad \langle \mathtt{swInit}, \emptyset \rangle$$

where $ta_0^{\mathtt{F}}$ is $\emptyset$.

4.2.8.4. The Calls by the Sweeper

The following rule formalizes the fact that when the type alist obtained by assuming false all the literals of a clause except the one which is being swept, the clause is trivially true and can be eliminated (see the arrow from $\mathit{Rth}_{\mathrm{SW}}$ to $\mathit{Rth}_{\mathrm{T}}$ in Figure 3).

$$\frac{\begin{array}{c} [h \vdash_{\mathrm{T}}^{\mathrm{ASM}} ta_{i-1}^{\mathtt{F}};\ l_i \to ta_i^{\mathtt{T}}, ta_i^{\mathtt{F}}, ar_i \mid 1 \leq i \leq n] \\ h \vdash_{\mathrm{SW}} db;\ \widetilde{cl} \leftarrow \widetilde{cl}' \end{array}}{h \vdash_{\mathrm{SW}} db;\ \widetilde{cl} \cup \{\{l_1, \ldots, l_n, l\}\} \leftarrow \widetilde{cl}'} \quad \langle \mathtt{swTcontr}, \{\mathtt{T} \in \{ar_1, \ldots, ar_n\}\} \rangle$$

where $ta_0^{\mathtt{F}}$ is $\emptyset$.

In NQTHM the sweeper invokes the rewriter in order to rewrite literals contained in the clauses which are being swept. The arrow from $\mathit{Rth}_{\mathrm{SW}}$ to $\mathit{Rth}_{\mathrm{R}}$ in Figure 3 denotes this invocation. The input is a polynomial database db, a type alist ta, and a literal l. The output is a literal l' (obtained rewriting l preserving propositional equivalence) and a set of linearization hypotheses $\widetilde{lh}$.

$$\frac{\begin{array}{c} [h \vdash_{\mathrm{T}}^{\mathrm{ASM}} ta_{i-1}^{\mathtt{F}};\ l_i \to ta_i^{\mathtt{T}}, ta_i^{\mathtt{F}}, ar_i \mid 1 \leq i \leq n] \\ h \vdash_{\mathrm{R}} ta_n^{\mathtt{F}}, db;\ l =_{\mathtt{B}} l' ; \widetilde{lh} \\ h \vdash_{\mathrm{SW}} db;\ \widetilde{cl} \cup \widetilde{cl}'' \cup split(\{l_1, \ldots, l_n\}, \widetilde{lh}) \leftarrow \widetilde{cl}' \end{array}}{h \vdash_{\mathrm{SW}} db;\ \widetilde{cl} \cup \{\{l_1, \ldots, l_n, l\}\} \leftarrow \widetilde{cl}'} \quad \langle \mathtt{swLit}, \emptyset \rangle$$

where $ta_0^{\mathtt{F}}$ is $\emptyset$, and $\widetilde{cl}''$ is $\mathit{splice}(\mathit{addhyps}(\{l_1,\ldots,l_n\},\widetilde{lh}\,),\mathit{ifclausify}(l'))$. The function

$$\mathit{splice} : \mathit{Cl} \times \mathrm{P}_\omega(\mathit{Cl}) \to \mathrm{P}_\omega(\mathit{Cl})$$

splices a set of clauses on a clause, i.e.

$$\mathit{splice}(cl,\widetilde{cl}) = \bigcup_{cl' \in \widetilde{cl}} \{cl \cup cl'\}.$$

The function

$$\mathit{ifclausify} : \mathit{Trm} \to \mathrm{P}_\omega(\mathit{Cl})$$

converts the input term to clausal form according to the `IF`-expressions present in it; if the term is `T` then $\emptyset$ is returned (and not $\{\{\mathtt{T}\}\}$), if it is `F` then $\{\emptyset\}$ is returned (and not $\{\{\mathtt{F}\}\}$).

4.2.8.5. The Calls by the Rewriter

The following two rules formalize the invocation of the linear arithmetic specialist by the rewriter (see the arrow from $\mathit{Rth}_{\mathrm{R}}$ to $\mathit{Rth}_{\mathrm{L}}$ in Figure 3).

$$\frac{h \vdash_{\mathrm{L}} ta;\ \langle db, \{\mathit{neg}(t)\}, \emptyset\rangle \to \langle db \cup \{p\}, \emptyset, \emptyset\rangle}{h \vdash_{\mathrm{R}} ta, db;\ t =_{\mathtt{B}} \mathtt{T}\,;\widetilde{lh}} \quad \langle\mathtt{rewT},\widetilde{c}\,\rangle$$

where $\widetilde{c}$ is the set of constraints

$$\begin{array}{l} \mathit{Impossible}(p) \\ p \sim \langle i, i_1 * t_1 + \cdots + i_n * t_n, \widetilde{lh}\,\rangle. \end{array}$$

$$\frac{h \vdash_{\mathrm{L}} ta;\ \langle db, \{t\}, \emptyset\rangle \to \langle db \cup \{p\}, \emptyset, \emptyset\rangle}{h \vdash_{\mathrm{R}} ta, db;\ t =_{\mathtt{B}} \mathtt{F}\,;\widetilde{lh}} \quad \langle\mathtt{rewF},\widetilde{c}\,\rangle$$

where $\widetilde{c}$ is the set of constraints

$$\begin{array}{l} \mathit{Impossible}(p) \\ p \sim \langle i, i_1 * t_1 + \cdots + i_n * t_n, \widetilde{lh}\,\rangle. \end{array}$$

The invocation of the typeset specialist by the rewriter is represented in Figure 3 by the two arrows from $\mathit{Rth}_{\mathrm{R}}$ to $\mathit{Rth}_{\mathrm{T}}$. In both cases, the input is a type alist and a term; the output is either a type set (if the invocation requires the computation of the typeset of the term) or two type alists and an assumption response (if instead the invocation requires assuming the term true or false). The following bridge rules describe the possible invocations of the typeset specialist by the rewriter.

The rules `recT` and `recF` respectively express the fact that a term $(r\ t)$, where r is a recognizer of a type $\underline{r}$ in the current history, is true or false under the current typeset assumptions.

$$\frac{h \vdash^{\mathrm{TS}}_{\mathrm{T}} ta;\ t \rightarrow \{\underline{r}\}}{h \vdash_{\mathrm{R}} ta, db;\ (r\ t) =_m \mathtt{T}\,;\emptyset} \quad \langle \mathtt{recT}, \emptyset \rangle$$

$$\frac{h \vdash^{\mathrm{TS}}_{\mathrm{T}} ta;\ t \rightarrow ts}{h \vdash_{\mathrm{R}} ta, db;\ (r\ t) =_m \mathtt{F}\,;\emptyset} \quad \langle \mathtt{recF}, \{ts \cap \{\underline{r}\} \sim \emptyset\} \rangle$$

The following rules are used to recursively rewrite the branches of an `IF`-expression.

$$\frac{\begin{array}{c} h \vdash^{\mathrm{ASM}}_{\mathrm{T}} ta;\ t \rightarrow ta, ta, \mathtt{T} \\ h \vdash_{\mathrm{R}} ta, db;\ t_1 =_m t_1' \,;\widetilde{lh} \end{array}}{h \vdash_{\mathrm{R}} ta, db;\ (\mathtt{IF}\ t\ t_1\ t_2) =_m t_1' \,;\widetilde{lh}} \quad \langle \mathtt{ifLeft}, \emptyset \rangle$$

$$\frac{\begin{array}{c} h \vdash^{\mathrm{ASM}}_{\mathrm{T}} ta;\ t \rightarrow ta, ta, \mathtt{F} \\ h \vdash_{\mathrm{R}} ta, db;\ t_2 =_m t_2' \,;\widetilde{lh} \end{array}}{h \vdash_{\mathrm{R}} ta, db;\ (\mathtt{IF}\ t\ t_1\ t_2) =_m t_2' \,;\widetilde{lh}} \quad \langle \mathtt{ifRight}, \emptyset \rangle$$

$$\frac{\begin{array}{c} h \vdash^{\mathrm{ASM}}_{\mathrm{T}} ta;\ t \rightarrow ta^{\mathtt{T}}, ta^{\mathtt{F}}, \mathtt{ANY} \\ h \vdash_{\mathrm{R}} ta^{\mathtt{T}}, db;\ t_1 =_m t_1' \,;\widetilde{lh}_1 \\ h \vdash_{\mathrm{R}} ta^{\mathtt{F}}, db;\ t_2 =_m t_2' \,;\widetilde{lh}_2 \end{array}}{h \vdash_{\mathrm{R}} ta, db;\ (\mathtt{IF}\ t\ t_1\ t_2) =_m (\mathtt{IF}\ t\ t_1'\ t_2') \,;\widetilde{lh}_1 \cup \widetilde{lh}_2} \quad \langle \mathtt{ifBranches}, \emptyset \rangle$$

The rule `if3` formalizes the third rewrite rule for `IF`.

$$\frac{h \vdash^{\mathrm{TS}}_{\mathrm{T}} ta;\ t \rightarrow \{\underline{\mathtt{T}}, \underline{\mathtt{F}}\}}{h \vdash_{\mathrm{R}} ta, db;\ (\mathtt{IF}\ t\ \mathtt{T}\ \mathtt{F}) =_m t\,;\emptyset} \quad \langle \mathtt{if3}, \emptyset \rangle$$

The rule `eq3` formalizes the first rewrite rule for `EQUAL`.

$$\frac{h \vdash^{\mathrm{TS}}_{\mathrm{T}} ta;\ t \rightarrow \{\underline{\mathtt{T}}, \underline{\mathtt{F}}\}}{h \vdash_{\mathrm{R}} ta, db;\ (\mathtt{EQUAL}\ t\ \mathtt{T}) =_m t\,;\emptyset} \quad \langle \mathtt{eq3}, \emptyset \rangle$$

The rule `npe` is used to rewrite to `F` an `EQUAL`-expression if its two arguments cannot be possibly equal.

$$\frac{\begin{array}{l} h \vdash_{\mathrm{T}}^{\mathrm{TS}} \mathit{ta};\ t_1 \to \mathit{ts}_1 \\ h \vdash_{\mathrm{T}}^{\mathrm{TS}} \mathit{ta};\ t_2 \to \mathit{ts}_2 \end{array}}{h \vdash_{\mathrm{R}} \mathit{ta}, \mathit{db};\ (\mathtt{EQUAL}\ t_1\ t_2) =_m \mathtt{F}\,;\emptyset} \quad \langle \mathtt{npe}, \{\mathit{Npe}(t_1, t_2, \mathit{ts}_1, \mathit{ts}_2)\}\rangle$$

The predicate

$$\mathit{Npe} \subseteq \mathit{Trm} \times \mathit{Trm} \times \mathit{Ts} \times \mathit{Ts}$$

consists of all the 4-tuples $\langle t_1, t_2, \mathit{ts}_1, \mathit{ts}_2\rangle$ such that certain conditions are verified, which are sufficient to guarantee that the two terms t_1 and t_2 cannot be equal (in the logic of NQTHM), provided ts_1 and ts_2 are their typesets (cf. [4] p. 121).

The following rules are used to return `T` or `F` from the rewriter if the assumed typeset of the term to be returned satisfies certain conditions, as explained in §2.

$$\frac{h \vdash_{\mathrm{T}}^{\mathrm{TS}} \mathit{ta};\ t \to \{\underline{\mathtt{T}}\}}{h \vdash_{\mathrm{R}} \mathit{ta}, \mathit{db};\ t =_m \mathtt{T}\,;\emptyset} \quad \langle \mathtt{retT}, \emptyset\rangle$$

$$\frac{h \vdash_{\mathrm{T}}^{\mathrm{TS}} \mathit{ta};\ t \to \{\underline{\mathtt{F}}\}}{h \vdash_{\mathrm{R}} \mathit{ta}, \mathit{db};\ t =_m \mathtt{F}\,;\emptyset} \quad \langle \mathtt{retF}, \emptyset\rangle$$

$$\frac{h \vdash_{\mathrm{T}}^{\mathrm{TS}} \mathit{ta};\ t \to \mathit{ts}}{h \vdash_{\mathrm{R}} \mathit{ta}, \mathit{db};\ t =_{\mathtt{B}} \mathtt{T}\,;\emptyset} \quad \langle \mathtt{retTB}, \{\mathit{ts} \cap \{\underline{\mathtt{F}}\} \sim \emptyset\}\rangle$$

4.2.8.6. The Calls by the Linear Arithmetic Specialist

The arrow from $\mathit{Rth}_{\mathrm{L}}$ to $\mathit{Rth}_{\mathrm{R}}$ in Figure 3 denotes the invocation of the rewriter by the linear arithmetic specialist in order to augment the current polynomial database. In our formalization the input is the current polynomial database and type alist. The output is a term to be linearized and a set of linearization hypotheses to be added to the linearization. The rule `augment` expresses this invocation.

$$\frac{\begin{array}{c} [h \vdash_{\mathrm{R}} \mathit{ta}, \mathit{db};\ \mathit{sbt}(l_i, \mathit{sb}) =_{\mathtt{B}} \mathtt{T}\,; \widetilde{\mathit{lh}}_{\,i} \bigm| 1 \leq i \leq n] \\ h \vdash_{\mathrm{R}} \mathit{ta}, \mathit{db};\ \mathit{sbt}(\mathit{lhs}, \mathit{sb}) =_{\mathtt{I}} \mathit{lhs}'\,; \widetilde{\mathit{lh}}_{\,n+1} \\ h \vdash_{\mathrm{R}} \mathit{ta}, \mathit{db};\ \mathit{sbt}(\mathit{rhs}, \mathit{sb}) =_{\mathtt{I}} \mathit{rhs}'\,; \widetilde{\mathit{lh}}_{\,n+2} \\ h \vdash_{\mathrm{L}} \mathit{ta};\ \langle \mathit{db}, \{(\mathtt{LESSP}\ \mathit{lhs}'\ \mathit{rhs}')\}, \emptyset\rangle \to \langle \mathit{db}, \emptyset, \{\widetilde{p}\,\}\rangle \end{array}}{h \vdash_{\mathrm{L}} \mathit{ta};\ \langle \mathit{db}, \widetilde{l}\,, \widetilde{\widetilde{p}}\,\rangle \to \langle \mathit{db} \cup \mathit{addlhyps}(\widetilde{p}\,, \bigcup_{i=1}^{n+2} \widetilde{\mathit{lh}}_{\,i}), \widetilde{l}\,, \widetilde{\widetilde{p}}\,\rangle} \quad \langle \mathtt{augment}, \widetilde{c}\,\rangle$$

where $\widetilde{c}$ is the set of constraints

$$\langle [l_i \bigm| 1 \leq i \leq n], (\mathtt{LESSP}\ \mathit{lhs}\ \mathit{rhs})\rangle \in \mathit{lr}{-}\mathit{rules}(h).$$

The function

$$\mathit{lr}{-}\mathit{rules} : H \to \mathrm{P}_{\omega}(\mathit{Lr})$$

returns the set of linear rules of a theory. (We assume there is a rule analogous to `augment` for linear rules whose conclusion is (`NOT` (`LESSP` t_1 t_2)).)

5. Conclusions and Future Work

As already mentioned, the logical component of NQTHM can be formalized as an OMRS in many different ways, i.e. by means of many different reasoning theories. For instance, using a reasoning theory closely resembling the formal system for the logic of NQTHM (as presented in [5]), would have been a perfectly legitimate choice. However, since our framework allows maximum flexibility in the definition of sequent systems and rules, we have preferred the reasoning theory presented in §4, both because its sequents and rules are "closer" to (the logical part of) the data structures and their elaborations in the actual implementation of the prover, and because it clearly exhibits the modularity of the prover. A major advantage of this choice is also that it can serve to give a clear and precise account of how, at the logical level, the 1979 version of NQTHM was enhanced by adding the linear arithmetic procedure: such an account can be given by first presenting a reasoning theory for the initial system and a reasoning theory for the stand-alone linear arithmetic procedure, then showing how they are modified and glued together (see [7]).

As we also mentioned at the beginning of §4, our framework allows a formalization of the logic component of NQTHM at the desired level of detail. The reasoning theory in §4 formalizes different parts of the system at different levels of detail (e.g. the typeset specialist is substantially treated as a black box, while the rewriter is quite detailed), in order to show the flexibility of the approach.

Future theoretical work for the logic component of OMRS will be mainly concerned with a refinement of the formalism to allow for reasoning theories to be parametric, and for defining the gluing of reasoning theories as their instantiation with actual parameters and then union of the obtained sequent systems and rules. In fact we think parameterization is a natural way to formally capture "openness" of systems, as different parameters may allow a same reasoning module to be integrated in different ways with different modules. Parameterization would also solve the current inadequacy about the introduction of "new" constraints for bridge rules, which we mentioned in a footnote in §3.1.

A theoretical framework has been also developed for the control component of OMRS [6], and used to formalize the inference strategies of the NQTHM simplifier [6]. A paper describing the control component of NQTHM as an OMRS is about to be written.

6. References

[1] A. Avron. Simple consequence relations. LFCS Report Series, Laboratory for the Foundations of Computer Science, Computer Science Department, University of Edinburgh, 1987.

[2] R. S. Boyer, M. Kaufmann, and J. S. Moore. The Boyer-Moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications*, (29):27 – 62, 1995.

[3] R. S. Boyer and Moore. J. S. Integrating decision procedures into heuristic theorem provers: A case study with linear arithmetic. Technical Report ICSCA-CMP-44, University of Texas at Austin, 1985. Also published in *Machine Intelligence 11*, Oxford University Press, 1988.

[4] Robert S. Boyer and J. Strother Moore. *A Computational Logic.* Academic Press, 1979.

[5] Robert S. Boyer and J. Strother Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[6] A. Coglio. "Definizione di un formalismo per la specifica delle strategie di inferenza dei sistemi di ragionamento meccanizzato e sua applicazione ad un sistema allo stato dell'arte", 1996. Master Thesis, DIST - University of Genoa (Italy). A short synthesis of the formalism is available in English, through the OMRS Web page at `http://www.mrg.dist.unige.it/omrs`.

[7] F. Giunchiglia, P. Pecchiari, and C. L. Talcott. Reasoning theories: Towards an architecture for open mechanized reasoning systems. Technical Report 9409-15, IRST, November 1994. A short version of this paper appears in the Proc. of the First International Workshop on Frontiers of Combining Systems (FroCoS'96), Munich, Germany, March 1996.

[8] Matt Kaufmann. Response to FM91 survey of formal methods: Nqthm and Pc-Nqthm. Technical Report 75, Computational Logic, Inc., March 1992.

[9] José Meseguer. General logics. In H.-D. Ebbinghaus et al., editor, *Logic Colloquium'87*, pages 275–329. North-Holland, 1989.