

The h1 Tool Suite

Jean Goubault-Larrecq

LSV/UMR 8643, CNRS, ENS Cachan & INRIA Futurs projet SECSI

61 avenue du président-Wilson, F-94235 Cachan Cedex

`goubault@lsv.ens-cachan.fr`

Phone: +33-1 47 40 75 68 Fax: +33-1 47 40 75 21

December 6, 2005

Abstract

This describes the theoretical basis and practical usage of the h1 tool suite. This is a set of tools that allow one to handle tree-regular languages in various formats, including deterministic, non-deterministic, and alternating finite tree automata, but also various fairly general clausal formats, the central one being the \mathcal{H}_1 class due to Nielson, Nielson, and Seidl. Alternatively, this can also be seen as a terminating automated theorem prover for the \mathcal{H}_1 class; or as an automated theorem prover for general clause sets, which however makes some controlled mistakes in the spirit of abstract interpretation: this is notably useful in proofs of security protocols. Other aspects of the h1 tool suite include producing certain forms of automated proofs by induction in the Coq proof assistant, deciding Presburger arithmetic, and displaying tree automata.

Contents

1	Introduction	2
2	Tree Automata, Clauses	6
3	The h1 Tool Suite through Examples	10
3.1	A Toy, Introductory Example	10
3.2	The Dreadsbury Mansion Murder Mystery Example	12
3.3	Computing with Tree Automata	14
3.3.1	Visualizing Tree Automata	16
3.3.2	Computing Intersections of Tree Automata	16
3.3.3	Checking Tree Automata for Emptiness, Testing Membership	19
3.3.4	Converting Alternating to Non-Deterministic Tree Automata	21
3.3.5	Purging Tree Automata	24

3.3.6	Determinizing Tree Automata	24
3.3.7	Computing Unions, Transitive Closures	29
3.3.8	Complete Deterministic Tree Automata, Taking Complements	35
3.4	The Needham-Schroeder Symmetric Key Protocol Example	39
4	The <code>h1</code> Prover	45
4.1	How to Use <code>h1</code>	45
4.2	Theoretical Background	50
4.3	Principle of Operation	50
5	Explaining and Checking Proofs: <code>h1trace</code>, <code>h1logstrip</code>	50
6	Model-Checking Clause Sets and Explaining the Absence of Proofs with <code>h1mc</code>	50
6.1	Theoretical Background	50
7	Determinizing Tree Automata with <code>pldet</code>	50
8	Converting XML Deterministic Tree Automata to Prolog Notation with <code>auto2pl</code>	50
9	Cleaning and Extracting Automata with <code>plpurge</code>	50
10	Converting Tree Automata and Prolog Programs to TPTP Files	50
11	Applying Morphisms to \mathcal{H}_1 Clause Sets with <code>tptpmorph</code>	50
12	Solving Presburger Arithmetic Formulas with <code>linauto</code>	50
13	Displaying Automata with <code>pl2gastex</code>	50
14	Log Files and <code>h1getlog</code>	52
15	Bugs	52
	Concept Index	54
	Command Index	55

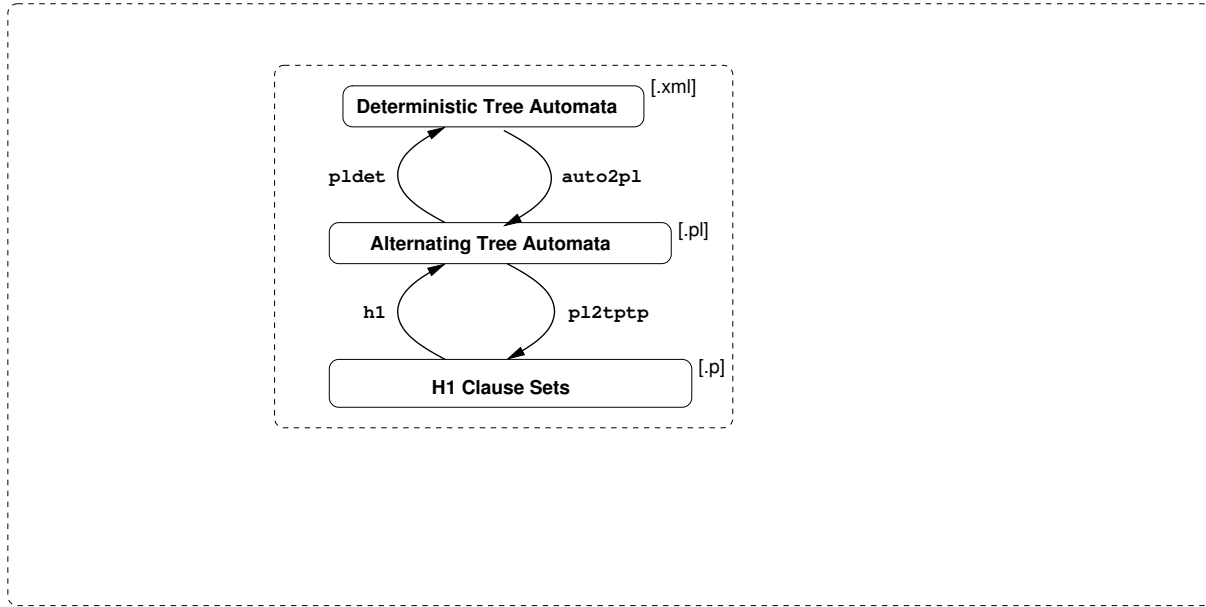
1 Introduction

The `h1` tool suite is a toolchest for handling finite tree automata, in various forms. There are basically three forms, from most constrained to least constrained:

1. as deterministic bottom-up tree automata;
2. as alternating tree automata;

3. as pure Prolog programs (sets of Horn clauses) in the \mathcal{H}_1 class [Nielson et al., 2002].

The global architecture is given in the following figure. Some things are still missing from it, and we will add them progressively later. The inner dashed box forms the *core* of the h1 tool suite. The outer dashed box is the h1 tool suite itself.



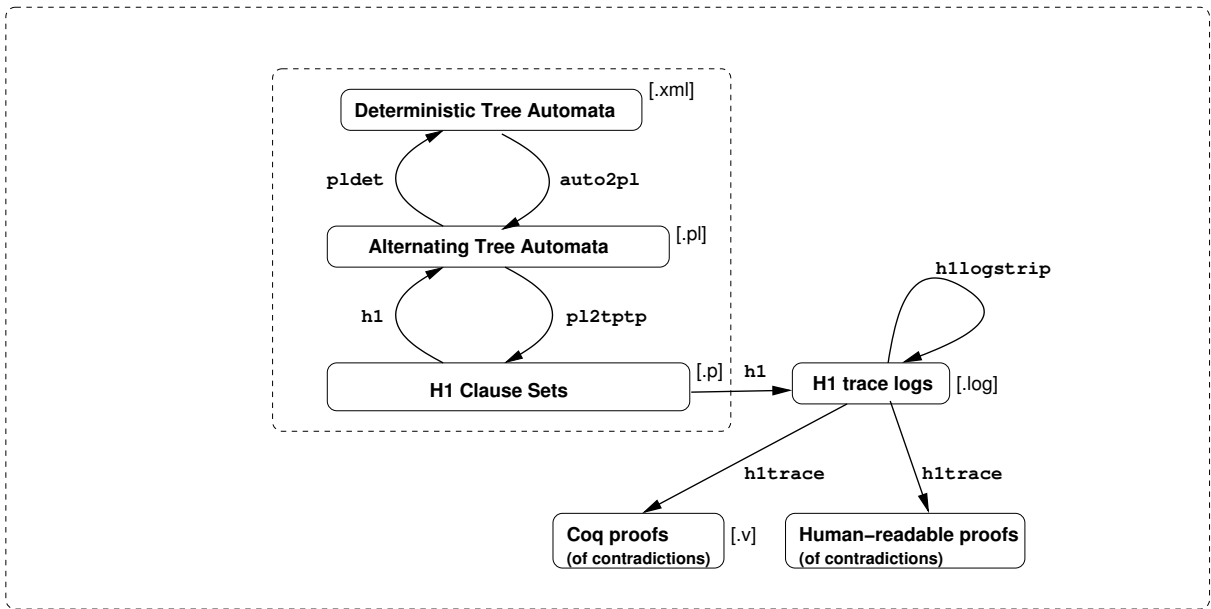
The main thing to understand is that the h1 tool suite includes tools to convert between all three formats of tree automata, forming the core of the h1 tool suite. The tools `h1`, `pldet`, `auto2pl`, `pl2tptp` are used to navigate between all three formats.

In the h1 tool suite, *deterministic tree automata* are represented in files in XML format.

Alternating tree automata are represented as Horn clauses, in Prolog notation; in particular, you can run them in any Prolog implementation (whatever the use of this may be).

Finally, \mathcal{H}_1 *clause sets* are represented in TPTP format. TPTP (a Thousand Problems for Theorem Provers) is a publicly available repository used to test automated theorem provers, due to Suttner and Sutcliffe [2002]. The h1 tool suite handles TPTP input files that contain only clauses. This allows you to use any automated, clausal theorem prover in place of the h1 prover if you so wish. The h1 prover, one of the tools of the h1 tool suite, is an automated theorem prover. In addition to searching for proofs, it is also able to produce counter-models and describe them as alternating tree automata; h1 is described in Section 4.

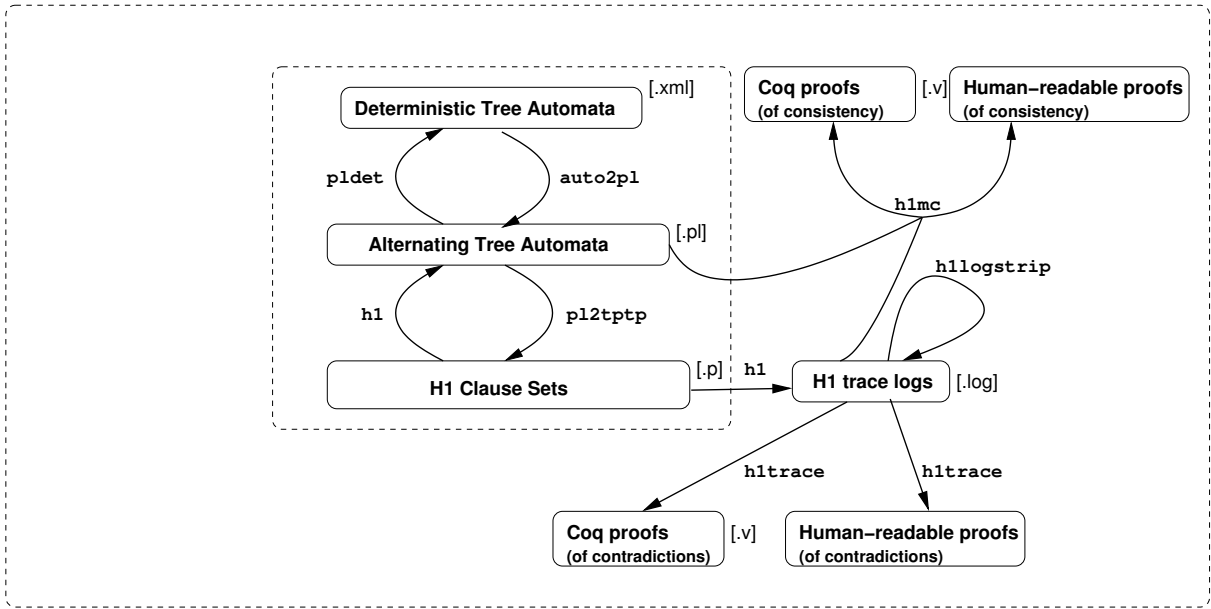
Additional tools operate on such counter-models and proofs. They form the rest of the h1 tool suite. As far as proofs are concerned, the h1 tool keeps a trace of all proof steps it did in a *trace log*. This trace log is in a proprietary format that the `h1trace` and `h1logstring` tools can work on. (Don't assume anything on this format, it may change in the future.) This is pictured in the following figure.



The purpose of `h1trace` is to explain the proofs found by `h1`, both to humans, as a proof in natural deduction in text format, and to machines, as a formal proof, in Coq notation. Coq is a proof assistant developed in the LogiCal team at INRIA Futurs [Barras et al., 1999–2003]. Since proof traces in `.log` format, as produced by `h1` and read by `h1trace` are huge, they are output by `h1` in `gzip`-compressed format. You may use the standard tool `zcat` in place of `cat` to read such compressed files. More in Section 5.

The `h1` tool is also able to work on general clause sets, not just in the \mathcal{H}_1 format. In this case, `h1` will output a proof candidate, which however may fail to be a proof. In case this is a wrong proof, `h1trace` will do its best to explain the wrong proof to humans. This can be used to design a true proof.

In case a prover does not find a proof of some given query, you usually have no choice but to trust it that there is indeed no proof. In case `h1` does not find a proof of some given query, it outputs a model. This model can be independently checked by the `h1mc` model-checking tool; `h1mc` takes as input both a description of the model \mathcal{M} , as an alternating tree automaton, and a trace log obtained from some given clause set S , and checks whether the clauses in S all hold in the model \mathcal{M} . This is a way of getting confident that there is indeed no proof of the initial query, i.e., that your query is wrong. More useful is the fact that `h1mc` can give you an explanation, based on \mathcal{M} , why your query is wrong. As with `h1trace`, this explanation can be made for humans, in text format, or in Coq format, to be checked independently by the Coq proof assistant. This is important in security protocols for instance, where a proof of secrecy consists in showing that there is no proof of the fact that the intruder can get hold of a given secret. This was shown by Selinger [2001], see Goubault-Larrecq [2004]. Adding `h1mc` gives you the following picture of the `h1` tool suite.



The `h1mc` tool is described in Section 6.

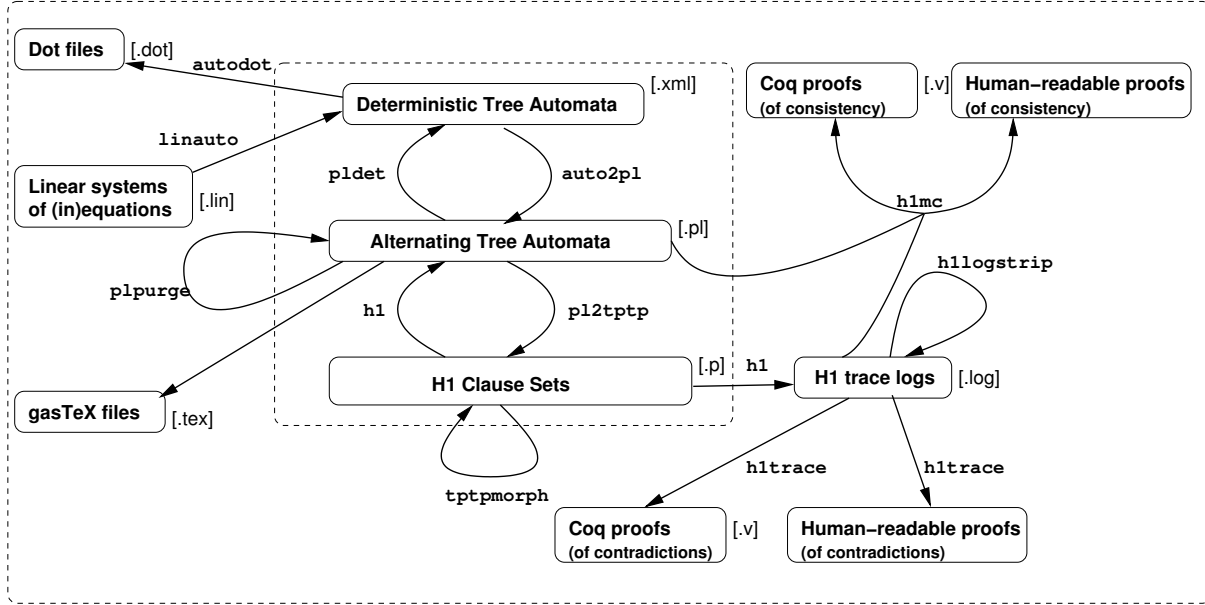
Additionally, various tools are provided. First, there are two conversion utilities, `auto2p1`, which converts deterministic tree automata in XML format into Prolog format (Section 8); and `p12tptp`, which converts alternating tree automata in Prolog format into TPTP format (Section 10). Since every deterministic tree automaton is a particular case of an alternating tree automaton, and every alternating tree automaton is a particular case of an \mathcal{H}_1 clause set (see Section 2), these two utilities entail no loss of information. In particular, to convert deterministic tree automata into \mathcal{H}_1 clause sets, just run `auto2p1` on the former, getting an intermediate file which you then convert to \mathcal{H}_1 clause sets using `p12tptp`.

Then, there are miscellaneous utilities. The first, `p1purge`, extracts the part of a tree automaton that is relevant to certain final states. In other words, it reads a tree automaton in Prolog format, and eliminates all states and transitions that do not reach any final state. This cleaning step is useful to help understand the structure of automata computed by `h1`, in particular as a preparation to calling `p12gastex`. The `p1purge` tool is described in Section 9. The second, `tptpmorph`, applies certain kinds of morphisms to languages represented as \mathcal{H}_1 clause sets. This is described in Section 11. The need for it should become apparent in conjunction with `linauto`, which implements Boudet and Comon [1996]’s algorithm for deciding quantifier-free Presburger formulae by converting them to automata. While `linauto` only deals with *quantifier-free* formulae, existential quantification can be dealt with by using `tptpmorph`, and universal quantification can be implemented using existential quantification and complementation. Complementation can be implemented by determinization (using `p1det`) and using `auto2p1` with the `-negate` option. The `linauto` tool is described in Section 12.

Finally, `p12gastex` is a utility that converts alternating tree automata to a sequence of `gasTEX` macros, which can then be used in conjunction with `LATEX` to display them on screen or include them in documents. A former attempt of a tree automaton visualizer is provided with `autodot`. The latter converts deterministic tree automata to files in `dot` format; `dot` can then be applied

to these files to display them graphically. This is only useful for relatively small automata, and is far from perfect for tree automata that are not just word automata. Don't expect too much from `pl2gastex` and `autodot` on anything else but small automata. The `autodot` utility is obsolescent: basically, it never gives satisfying output. The `pl2gastex` utility is more satisfactory, but not perfect. See Section 13.

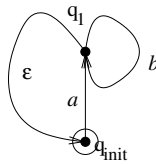
At this point the picture of the `h1` tool suite is complete—up to the fact that I did not talk about some minor extensions yet, and that more extensions may be added in the future.



2 Tree Automata, Clauses

Automata on finite words are a very classical data structure to represent sets of words—possibly infinite sets. Not all sets of words can be described this way; such special sets are called *regular* languages. Regular languages are reasonably expressive, and all the basic operations are computable on them: testing for membership, for vacuity, computing unions, intersections, complements notably.

An example of automaton is shown below. This is just a directed graph. Vertices are traditionally called *states*, and edges are called *transitions*. Two other components are needed. First, we need an *initial state*: in the example, we assume this is q_{init} . Second, we need *final states*. The convention is that final states are circled; here, the only final state is q_{init} .



Let us play a game. You start at the initial state q_{init} , then you must follow some transitions (whichever you wish, you may even repeat the same transition as long as you wish), until you reach a final state. While traveling around, collect the letters that label the transitions: this gives you a sequence of letters, i.e., a *word*. (One exception, though: the ϵ symbols means “no letter”, and you should just go through the corresponding transition without collecting any letter at all.) The fact that you reached the final state is how you decide that the automaton *recognized* the word.

For example, starting from q_{init} , you may go up the a transition, then come back along the ϵ transition. Since you’ve reached a final state, namely q_{init} (which is both initial and final), the word a is recognized by the automaton. Or you may have decided to start from q_{init} , go up the a transition, turn once inside the b loop, then come back through the ϵ -transition: the word ab is recognized. In fact, you might have looped as many times as you wished, so any word ab^n , $n \in \mathbb{N}$, is recognized, too.

There is no need to stop when you reach a final state. For example, we may travel along a , then b twice, then along ϵ , then again along a , then b three times, then ϵ , showing that ab^2ab^3 is recognized. While we are at it, since q_{init} is already final, the empty word (with no letter) is also recognized. We write ϵ for the empty word, as is traditional.

To wrap up the example, the above automaton recognizes exactly the language $(ab^*)^*$, that is the set of all words on the alphabet $\{a, b\}$ which are concatenations of words ab^n , $n \in \mathbb{N}$. It turns out that this is just the set of words that, if non-empty, start with a .

Alternatively, we can describe this same set of words as a set of Horn clauses, i.e., as a very simple Prolog program. To this end, create a fresh predicate symbol q for each state q . The meaning of $q(t)$ is that the word t should be *recognized at* q , i.e., there is a trip along the transitions in the automaton, starting from the initial state, and ending exactly on the state q , along which the letters collected form the word t . We also encode words as terms: ϵ will be a constant denoting the empty word, and we add each letter ℓ at the end of word t by writing the term $\ell(t)$; this means in particular that each letter is now viewed as a unary function symbol. The automaton above then gets described as a Prolog program with one clause for each transition, plus one to say that the empty word is recognized at the initial state:

$$\begin{array}{ll} q_{init}(\epsilon) & q_1(a(X)):-q_{init}(X) \\ q_1(b(X)):-q_1(X) & q_{init}(X):-q_1(X) \end{array}$$

You may ask Prolog whether the word ab^2 is recognized at q_{init} by submitting the query

$$?q_{init}(b(b(a(\epsilon))))$$

and it will answer “yes”. Prolog will also answer the query

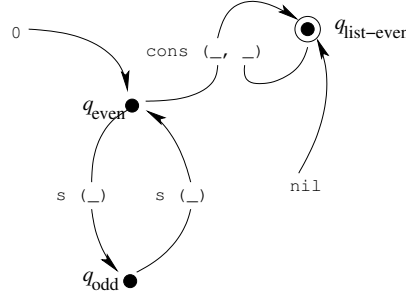
$$?q_{init}(b(b(a(b(\epsilon)))))$$

by “no”, meaning that bab^2 is not in the language.

Prolog, as a notation, is fine, as we shall see. Prolog, as a tool to check properties, is ill-suited: on the more complex clauses we shall encounter below, Prolog would loop infinitely in general; h1 on the other hand is meant to return, always, on its input clauses.

The notion of automata, on words, can be generalized to automata on first-order terms, a.k.a., *tree automata*. These are very similar structures, except they recognize sets of ground terms. The automaton of Figure 1, to take an example, recognizes the set of all lists of even natural numbers at state $q_{\text{list-even}}$. To be precise, it recognizes the set of all terms $\text{cons}(t_1, \text{cons}(t_2, \dots, \text{cons}(t_n, \text{nil}) \dots))$, where each t_i is of the form $S^{n_i}(0)$, n_i even. Note that the transition 0 (up left) starts from no state, while the transition $\text{cons}(-, -)$ (middle) starts from a pair of states, q_{even} and $q_{\text{list-even}}$.

Figure 1: Recognizing the lists of even natural numbers



To define the semantics of tree automata, the simplest is just to describe their translation to Horn clauses. Each transition again gives rise to exactly one clause:

$$\begin{array}{lll}
 q_{\text{even}}(0) & q_{\text{even}}(S(X)):-q_{\text{odd}}(X) & q_{\text{odd}}(S(X)):-q_{\text{even}}(X) \\
 q_{\text{list-even}}(\text{cons}(X, Y)):-q_{\text{even}}(X), q_{\text{list-even}}(Y) & q_{\text{list-even}}(\text{nil}) &
 \end{array}$$

There is no need to define initial states in tree automata; e.g., 0 is recognized at q_{even} , using the transition $q_{\text{even}}(0)$, of arity 0.

In general, a *tree automaton* is any finite set S of clauses of the form

$$P(f(X_1, \dots, X_n)) \quad :- \quad P_1(X_1), \dots, P_n(X_n) \quad (1)$$

where X_1, \dots, X_n are pairwise distinct variables. When $n = 0$, we retrieve initial clauses such as $q_{\text{even}}(0)$. When n is restricted to be at most 1, tree automata are just ordinary, word automata (without ϵ -transitions).

A set of ground terms is called *regular* if and only if it is exactly the set of terms t such that t is recognized at P in some tree automaton S (i.e., such that $P(t)$ follows logically from the clauses in S).

The format of clauses (1) is very particular. First, clauses (1) are definite clauses. Formally, *definite clause* are implications of the form

$$P(t):-P_1(t_1), \dots, P_n(t_n) \quad (2)$$

where $P(t), P_1(t_1), \dots, P_n(t_n)$ is an unordered set of atoms. (An *atom* is just a predicate P applied to some term t ; read “ P holds of t ”, or “ t is recognized at P ”.) If $n = 0$, this is called a *fact*, and is often written just $P(t)$.

Any set S of definite clauses has a *least Herbrand model*. A *Herbrand model* is just a collection of ground atoms $P(t)$. The least Herbrand model $\mathcal{H}(S)$ of S can be described as follows. First, it contains all ground instances of the facts in S . Then, while there is a ground instance $P(t\sigma) :- P_1(t_1\sigma), \dots, P_n(t_n\sigma)$ of a clause $P(t) :- P_1(t_1), \dots, P_n(t_n)$ in S , and $P_1(t_1\sigma), \dots, P_n(t_n\sigma)$ are in $\mathcal{H}(S)$, then add $P(t\sigma)$ to $\mathcal{H}(S)$. This procedure does not terminate in general, but enumerates $\mathcal{H}(S)$.

In particular, any tree automaton has a least Herbrand model. It turns out that the ground atom $P(t)$ is in the least Herbrand model $\mathcal{H}(S)$ if and only if t is recognized at P in the tree automaton S . Therefore, we may generalize the notion of a term t being recognized at some predicate P in any set of definite clauses, by requiring that $P(t)$ is in $\mathcal{H}(S)$.

A *goal clause* is an implication of the form

$$\perp :- P_1(t_1), \dots, P_n(t_n) \quad (3)$$

where \perp is meant to denote false. A *Horn clause* is a definite or a goal clause. We shall also write $P(t)$ instead of $P(t) :-$ when $n = 0$, and \perp instead of $\perp :-$.

Not any set S of Horn clauses has a model. If it has one, that is, if S is *satisfiable*, then it again has a least Herbrand model $\mathcal{H}(S)$. Again, we say that t is recognized at P in S if and only if $P(t)$ is in $\mathcal{H}(S)$.

Then it can be shown that the ground term t is recognized at P in the set S of definite clauses if and only if S plus the goal clause $\perp :- P(t)$ is *unsatisfiable* (i.e., not satisfiable); that P is *empty* in S , i.e., that P recognizes no term in S , if and only if S plus the clause $\perp :- P(X)$ is unsatisfiable, where X is a variable. And there are automated means, called automated theorem provers, to check the unsatisfiability of clause sets. Unfortunately, they do not always terminate. The `h1` tool always terminates, but only deals with so-called \mathcal{H}_1 clauses [Nielson et al., 2002].

We have said above that tree automata clauses (1) were very particular, because they were definite clauses, and in particular they always have a least Herbrand model. They are also particular in that the *head* (the atom at the left of $:-$) is restricted to be of the form $P(t)$ with t itself of the form $f(X_1, \dots, X_n)$, where X_1, \dots, X_n must be distinct variables; and in that there is no function symbol at all in the *body* (the set of atoms at the right of $:-$).

If you do not restrict the form of Horn clauses (2) and (3), then any prover that operates on them is forced to relinquish either termination, soundness, or completeness. This is because the satisfiability of Horn clauses is *undecidable* [Devienne et al., 1996]. Most automated theorem provers in existence are *sound*, i.e., if they deduce a contradiction from S then S is unsatisfiable, and *complete*, i.e., if S is unsatisfiable then they can derive a contradiction from S . Therefore they have to fail to terminate sometimes. On the other hand, `h1` is complete and terminates, but is only sound on the subset of so-called \mathcal{H}_1 -clauses.

The \mathcal{H}_1 -clauses are exactly the Horn clauses, except that definite clauses (1) are restricted to have a head of the form $P(X)$, where X is a variable, or $P(f(X_1, \dots, X_n))$, where X_1, \dots, X_n are distinct variables. While this is not the definition Nielson et al. [2002], this is equivalent to it, see Goubault-Larrecq [2005].

On general Horn clauses, `h1` applies an abstraction function which makes it still a terminating and complete prover, but one which is unsound in general. So you might want to see `h1` as a

counter-model finder rather than a prover. In some cases, though, `h1` produces unsound “proofs” that may be indicative of actual proofs.

Remember that `h1` is sound, complete, and terminating on \mathcal{H}_1 clauses. Note that automata clauses are a special case of \mathcal{H}_1 clauses. In fact, when `h1` terminates, starting from a set S of \mathcal{H}_1 clauses, there are two possible outcomes:

- a contradiction has been derived; then S is unsatisfiable.
- no contradiction was derived; then S is satisfiable.

In the latter case, `h1` also produces a model of S , in the form of an *alternating tree automaton*, i.e., a set of clauses of the form

$$P(f(X_1, \dots, X_n)) \quad :- \quad B_1(X_1), \dots, B_n(X_n) \quad (4)$$

where $B_i(X_i)$ is a *block*, i.e., a list of atoms $P_{i1}(X_i), \dots, P_{in_i}(X_i)$. Note that tree automata clauses are a special case of alternating tree automata clauses (take $n_i = 1$ for each i), while alternating tree automata clauses are special \mathcal{H}_1 clauses (with no function symbol in the body, the head is of the form $P(f(X_1, \dots, X_n))$ and not $P(X)$, and every variable free in the body is among X_1, \dots, X_n).

From a theoretician’s perspective, this means that \mathcal{H}_1 clauses are not more expressive than alternating tree automata clauses. And it is well-known that alternating tree automata are not more expressive than plain tree automata: the languages defined by satisfiable \mathcal{H}_1 clause sets are just, again, the regular tree languages. However, \mathcal{H}_1 offers considerably more freedom in describing such languages than just using tree automata, because of the general form of \mathcal{H}_1 clauses.

The paper that introduced \mathcal{H}_1 is Nielson et al. [2002]. For some background theory on the way `h1` works, deciding \mathcal{H}_1 and converting \mathcal{H}_1 clause sets to tree automata by resolution techniques, and abstracting general clause sets to \mathcal{H}_1 clause sets, see Goubault-Larrecq [2005].

3 The `h1` Tool Suite through Examples

The `h1` prover is the core of the `h1` tool suite, and we shall explain the tool suite by running `h1` on several examples.

The `h1` prover is invoked by calling `h1` with a sequence of flags, ended by a file name. The file name should contain a set of clauses in TPTP clause format. Such files conventionally end with the `.p` extension—but there is no obligation. Also, giving a single dash `-` as file name forces `h1` to read the input clause set from standard input.

3.1 A Toy, Introductory Example

Examples are given in the distribution package. Here is a very small one (file `test1.p`):

```
|| input_clause ( clause1 , conjecture ,
||               [ ++p(a) ] ).
```

```

input_clause (clause2 , conjecture ,
               [---p(X) , ++p(f(X))]).
input_clause (clause3 , conjecture ,
               [---p(f(f(X)))]).

```

This contains three clauses. Each clause is introduced by the keyword **input_clause**. The first argument, clause1, clause2, or clause3 above, is the name of the clause. Names are used in sundry ways, mainly for explanation and documentation purposes. It is good practice to give each clause a different name, but the tools of the h1 tool suite should work even when several clauses have the same name.

The second argument can be the keyword `conjecture` or `axiom`; h1 just does not care: write what you prefer here.

Finally, the third argument, enclosed between square brackets, is the clause itself. It is a list of literals, separated by commas. Each literal starts with a sign, ++ for positive literals, -- for negative literals. The clauses above are, in a more traditional notation:

$$\begin{aligned}
 & p(a) \\
 & \neg p(X) \vee p(f(X)) \\
 & \neg p(f(f(X)))
 \end{aligned}$$

This example is in fact a set of Horn clauses, which would be written in Prolog notation:

$$\begin{aligned}
 & p(a) \\
 & p(f(X)) \text{ :- } p(X) \\
 & \perp \text{ :- } p(f(f(X)))
 \end{aligned}$$

Launch h1 on this file, test1.p, by typing

```
h1 test1.p
```

You will get the answer

```
*** Derived: clause3 ***
```

which means that a contradiction was found, using the clause named clause3 in the last step. In other words, in the present example, it says that a fact of the form $p(f(f(t)))$, matching the body of clause3, can be deduced from the definite clauses (here, clause1 and clause2)

So far, so good, this is typically the least you could expect from a theorem prover.

However, h1 also produced two more files, test1.log.gz and test1.model.pl. If you're curious, look at test1.log.gz, by running

```
zcat test1.log.gz
```

Oh well, it is long, but it is not meant to be read by a human reader! If you're perspicuous enough, you'll find some meaning buried inside this. However, this is really meant as a log file, from which h1trace can extract a (mostly) readable proof (Section 5), and which h1mc can use to get some essential information it needs (Section 6).

3.2 The Dreadsbury Mansion Murder Mystery Example

Here is a more complicated example, due to Len Schubert. This is problem 55 of Pelletier [1986].

Someone in Dreadsbury Mansion killed Aunt Agatha. Agatha, the butler, and Charles live in Dreadsbury Mansion, and are the only ones to live there. A killer always hates, and is no richer than his victim. Charles hates noone that Agatha hates. Agatha hates everybody except the butler. The butler hates everyone not richer than Aunt Agatha. The butler hates everyone whom Agatha hates. Noone hates everyone. Who killed Agatha?

The problem is formalized in file `butler-puzzle.p`. The clauses are as follows. First, “Agatha, the butler, and Charles live in Dreadsbury Mansion, and are the only ones to live there.”:

```
agatha_in_mansion    in_mansion(agatha)
charles_in_mansion    in_mansion(charles)
butler_in_mansion     in_mansion(butler)
```

Then, “A killer always hates, and is no richer than his victim.”, that is, if X killed Y , then it must be the case that X hates Y , and on the other hand that X is not richer than Y :

```
killer_hates_victim    hates( $X, Y$ ) :- killed( $X, Y$ )
killer_no_richer       not_richer( $X, Y$ ) :- killed( $X, Y$ )
```

Next, “Charles hates noone that Agatha hates.” In other words, it is impossible that Charles and Agatha hate the same person X :

```
charles_hates_noone_agatha_hates   $\perp$  :- hates(charles,  $X$ ), hates(agatha,  $X$ )
```

To write “Agatha hates everybody except the butler.”, we just say that Agatha hates herself and Charles:

```
Agatha_hates_herself    hates(agatha, agatha)
Agatha_hates_charles     hates(agatha, charles)
```

Now, “The butler hates everyone not richer than Aunt Agatha.”:

```
butler_hates_everyone_not_richer_than_agatha  hates(butler,  $X$ ) :- not_richer( $X$ , agatha)
```

Then, “The butler hates everyone whom Agatha hates.”:

```
butler_hates_everybody_agatha_hates    hates(butler,  $X$ ) :- hates(agatha,  $X$ )
```

Next, “Noone hates everyone.”, which we formulate as “noone hates Agatha, Charles, and the butler”:

```
noone_hates_everyone   $\perp$  :- hates( $X$ , agatha), hates( $X$ , butler), hates( $X$ , charles)
```

Finally, we explore who may have killed Aunt Agatha. To do this, we shall enumerate the potential murderers, and use the C preprocessor `cpp` to replace the macro identifier `WHO` in the following clause by each resident of Dreadsbury Mansion:

```
WHO_killed_agatha killed(WHO,agatha)
```

Let us test whether the butler killed Agatha. Run `butler-puzzle.p` through `cpp` with `WHO` equal to `butler`, and feed the output to `h1`; `h1` reads from standard input when given `-` as file name:

```
cpp -P -DWHO=butler butler-puzzle.p | h1 -
```

You get the output:

```
*** Derived: noone_hates_everyone ***
```

In other words, the butler cannot have killed Agatha (contrarily to proper conventions in popular whodunnit mysteries), because this would contradict the fact that noone hates everyone.

More information can be gotten from the trace file `h1out.log.gz`. (Running `h1` on file `<file>.p` produces a trace file `<file>.log.gz`. If no input file is given, as here, the trace file is called `h1out.log.gz`.) We shall explain how to use this trace file in Section 5. For now, just run

```
zcat h1out.log.gz | h1trace - >dummy
```

and open the file `dummy`. You'll see that the first lines say:

```
. #false(noone_hates_everyone) [noone_hates_everyone].
using assumption #false(noone_hates_everyone) :-
    hates(X1,charles), hates(X1,butler), hates(X1,agatha).
{X1=butler}
```

In other words, assuming the butler killed Agatha would involve that the butler hates everyone, which is impossible. The rest of file `dummy` is a tree-like proof that indeed the butler hates everyone, i.e., hates Agatha, Charles, and himself in this case.

So did Charles kill Agatha instead?

```
cpp -P -DWHO=charles butler-puzzle.p | h1 -
```

No, this would contradict the fact that Charles hates noone that Agatha hates. In this case, Charles hates Agatha, and Agatha hates herself, whence the contradiction.

```
*** Derived: charles_hates_noone_agatha_hates ***
```

There is only one possibility remaining: that Agatha killed herself.

```
cpp -P -DWHO=agatha butler-puzzle.p | h1 -
```

and indeed, `h1` does not complain: assuming Agatha killed herself leads to no contradiction. The file `h1out.model.pl` describes the least model of this clause set.

This could have been found automatically by a simple `sh` script, listing all possible murderers of Aunt Agatha.

```

for who in butler charles agatha
do
  if (cpp -P -DWHO=$who butler-puzzle.p | h1 - 2>&1\
                                         | grep -q Derived)
  then echo $who did not kill agatha.
  else echo $who may have killed agatha.
  fi
done

```

Since somebody killed Agatha, it must be herself.

```

butler did not kill agatha.
charles did not kill agatha.
agatha may have killed agatha.

```

The point of this example is that, first, the clauses are clearly not (alternating tree) automata clauses; and second, that they are \mathcal{H}_1 clauses. Check this by running `h1` with the `-check-h1 2` option:

```
cpp -P -DWHO=agatha butler-puzzle.p | h1 -check-h1 2 -
```

This makes `h1` run as above, except it would have failed if any of the input clauses were not in \mathcal{H}_1 . By default, `h1` runs as `h1 -check-h1 0`, which does not check anything, but computes an approximation, see Section 3.4.

That the clauses of `butler-puzzle.p` are in the \mathcal{H}_1 class may seem surprising. After all, \mathcal{H}_1 clauses are required to use only *unary* predicates, i.e., all predicate letters P can only take one argument. This is certainly not the case of the `hates`, `killed`, and `not_richer` predicates above, which are all binary!

The trick here is that, when `h1` sees an n -ary predicate $P(t_1, \dots, t_n)$ in its input, it converts it first to $P(f_P(t_1, \dots, t_n))$ for some fresh n -ary function symbol f_P . (Different occurrences of P correspond to the same symbol f_P .) This makes P unary, and does not change the semantics of clauses in any essential way. Under the hood, `h1` typically builds f_P by prepending a sharp sign `#` in front of the name of P , guaranteeing that no clash occurs with any function symbol you may have used. Don't count on it, though, as this may change in future releases. Also, the `h1` tools try to hide this kludge as much as they can, and will happily parse and print n -ary predicate symbols.

We shall return to this example in Section 3.3.8.

3.3 Computing with Tree Automata

Assume that we wish to compute the intersection of the languages \mathcal{L}_1 of all lists of even natural numbers, and \mathcal{L}_2 of all trees with binary nodes labeled with *cons*, and whose leaves are either *nil* or natural numbers of the form $3n + 2$, $n \in \mathbb{N}$.

We have already seen what an automaton recognizing \mathcal{L}_1 looked like, see Figure 1. In TPTP format, this is file `listeven.p`:

```

input_clause(o_even , axiom ,
              [++even(o)]).

input_clause(suc_even_odd , axiom ,
              [--even(X) , ++odd(s(X))]).

input_clause(suc_odd_even , axiom ,
              [--odd(X) , ++even(s(X))]).

input_clause(nil_even_list , axiom ,
              [++list_even(nil)]).

input_clause(cons_even_list , axiom ,
              [--even(X) , --list_even(Y) ,
               ++list_even(cons(X,Y))]).

```

The language \mathcal{L}_2 is described by the predicate (final state) `tree_3n_plus_2`. Look at file `tree3plus2.p`:

```

input_clause(o_zero_mod_3 , axiom ,
              [++zero_mod_3(o)]).

input_clause(suc_zero_one_mod_3 , axiom ,
              [--zero_mod_3(X) , ++one_mod_3(s(X))]).

input_clause(suc_one_two_mod_3 , axiom ,
              [--one_mod_3(X) , ++two_mod_3(s(X))]).

input_clause(suc_two_zero_mod_3 , axiom ,
              [--two_mod_3(X) , ++zero_mod_3(s(X))]).

input_clause(nil_3n_plus_2_tree , axiom ,
              [++tree_3n_plus_2(nil)]).

input_clause(two_mod_3_3n_plus_2_tree , axiom ,
              [--two_mod_3(X) , ++tree_3n_plus_2(X)]).

input_clause(cons_3n_plus_2_tree , axiom ,
              [--tree_3n_plus_2(X) , --tree_3n_plus_2(Y) ,
               ++tree_3n_plus_2(cons(X,Y))]).

```

By the way, we can convert any automaton in TPTP format into Prolog format by running `h1` with the `-no-trim` option:

```
h1 -no-trim tree3plus2.p
```

Normally, `h1`'s default is to use the `-trim` option, which trims away all clauses that are obviously not needed for deriving a contradiction. (See Section 4.1 for more information on `-trim` and `-no-trim`.) In this case, trimming would just eliminate all clauses! Since we are not looking for a contradiction, we just run `h1` without trimming, and get a model in `tree3plus2.model.pl`:

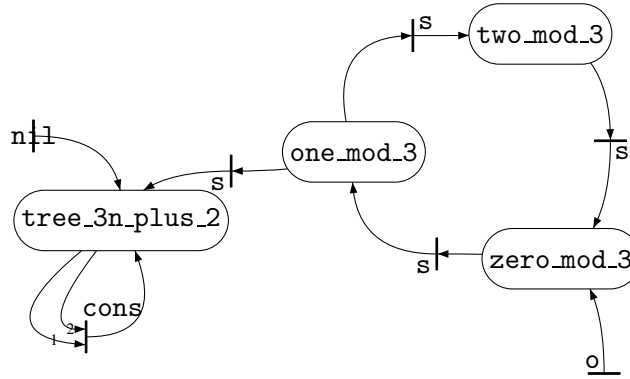
```
one_mod_3(s(X)) :- zero_mod_3(X).
tree_3n_plus_2(nil).
tree_3n_plus_2(s(X)) :- one_mod_3(X).
tree_3n_plus_2(cons(X1,X2)) :- tree_3n_plus_2(X1), tree_3n_plus_2(X2).
two_mod_3(s(X)) :- one_mod_3(X).
zero_mod_3(o).
zero_mod_3(s(X)) :- two_mod_3(X).
```

In other words, `h1` can be used to convert any set of \mathcal{H}_1 clauses into an equivalent alternating tree automaton by running it with the `-no-trim` option and looking into the generated model file, ending in `.model.pl`.

3.3.1 Visualizing Tree Automata

Before we compute the intersection of \mathcal{L}_1 and \mathcal{L}_2 , let us visualize the automaton defining \mathcal{L}_2 . This is accomplished using `pl2gastex`, see Figure 2. For more information on `pl2gastex`, and how to read such pictures precisely, see Section 13.

Figure 2: Trees with leaves equal to `nil` or to $3n + 2$, $n \in \mathbb{N}$



3.3.2 Computing Intersections of Tree Automata

Now compute the intersection. Build a file, say `list_even_inter_tree3plus2.p`, by concatenating the clauses from `listeven.p` and from `tree3plus2.p`, and add the so-called

intersection clause

$$q(X) \text{ :- list_even}(X), \text{tree_3n_plus_2}(X)$$

meaning that q holds of all terms that are both lists of even numbers, and trees as recognized at `tree_3n_plus_2`. Just run the following commands:

```
OUT=list_even_inter_tree3plus2
cat listeven.p tree3plus2.p >$OUT.p
echo "input_clause ($OUT, axiom, \
   [++q(X), --list_even(X), \
    --tree_3n_plus_2(X)])" >>$OUT.p
```

Because of the intersection clause above, the resulting clause set is not an alternating tree automaton as we have defined it. However, run `h1 -no-trim` on it:

```
h1 -no-trim list_even_inter_tree3plus2.p
```

and look at the generated alternating tree automaton. This is obtained, as usual, in a file named `list_even_inter_tree3plus2.model.pl`:

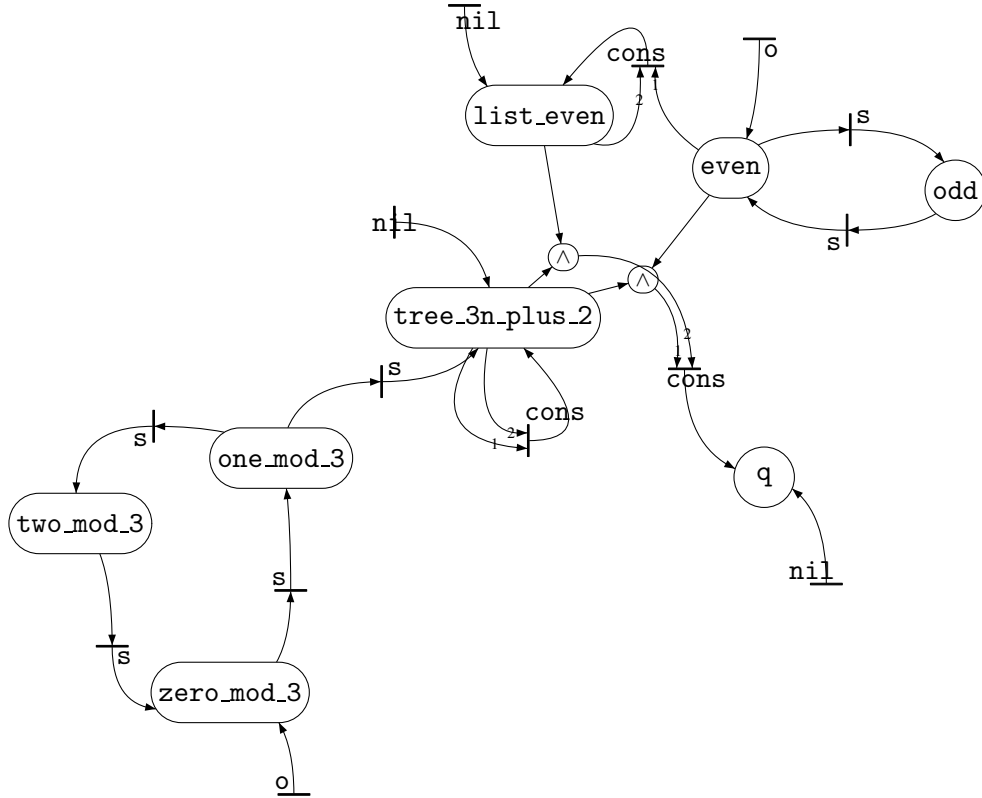
```
q(nil).
q(cons(X1,X2)) :- tree_3n_plus_2(X1), even(X1), tree_3n_plus_2(X2),
    list_even(X2).
two_mod_3(s(X)) :- one_mod_3(X).
odd(s(X)) :- even(X).
zero_mod_3(o).
zero_mod_3(s(X)) :- two_mod_3(X).
one_mod_3(s(X)) :- zero_mod_3(X).
tree_3n_plus_2(nil).
tree_3n_plus_2(s(X)) :- one_mod_3(X).
tree_3n_plus_2(cons(X1,X2)) :- tree_3n_plus_2(X1), tree_3n_plus_2(X2).
list_even(nil).
list_even(cons(X1,X2)) :- even(X1), list_even(X2).
even(o).
even(s(X)) :- odd(X).
```

Graphically, this is the alternating tree automaton of Figure 3.

Note the presence of new nodes labeled \wedge in Figure 3. They represent intersections of languages. Indeed, there are two ways one can construct an element of q . First, there is *nil*. Second, there are terms of the form *cons* applied to two arguments X_1 and X_2 , where X_1 is recognized *both* at `tree_3n_plus_2` and at `even`, and X_2 is recognized *both* at `tree_3n_plus_2` and at `list_even`.

The alternating tree automaton above recognizes (at q) the terms in the intersection of \mathcal{L}_1 and \mathcal{L}_2 . One should observe that computing intersections of two languages by concatenating the clause sets defining each and adding an intersection clause is rather cavalier. It is only correct here because

Figure 3: Trees with leaves equal to nil or to $3n + 2$, which are lists of even natural numbers at the same time



the two files `listeven.p` and `tree3plus2.p` share no predicate symbol. In general, one might allow shared predicate symbols, provided they have the same semantics in each file. For example, it is legal for the two files to both use the predicate `even` provided it denotes the set of even natural numbers in both. Otherwise, strange things may happen (an over-approximation will be computed).

3.3.3 Checking Tree Automata for Emptiness, Testing Membership

It may not be completely obvious whether such an alternating tree automaton is empty or not. (To say the least. The problem is DEXPTIME-complete.) Let us see whether the intersection state q is empty or not. In general, given a satisfiable set S of Horn clauses (e.g., definite clauses, in particular alternating tree automata clauses), the language of terms recognized at state P in S is empty if and only if S plus the clause $\perp:-P(X)$ is satisfiable. Running:

```
(cat list_even_inter_tree3plus2.p;\
echo "input_clause(q_is_not_empty, conjecture, [--q(X)]).")\
| h1 -log-out - >list_even_inter_tree3plus2.log
```

yields

```
*** Derived: q_is_not_empty ***
```

meaning that there are indeed terms recognized at state q in the intersection.

We have kept a trace of the derivation in the log file `list_even_inter_tree3plus2.log`. We can then use `hltrace` to get a mostly readable proof of the fact that q is non-empty; in particular, to have an example of a term recognized at q :

```
%*-mode:outline;outline-regexp:"[0-9a-z.]+"*-
. #false(q_is_not_empty).
  using assumption #false(q_is_not_empty) :- q(X1).
    {X1=nil}
1. q(nil).
  using assumption q(X1) :- tree_3n_plus_2(X1), list_even(X1).
    {X1=nil}
1.1. tree_3n_plus_2(nil) by assumption.
1.2. list_even(nil) by assumption.
```

In fact, the empty list *nil* is recognized at q ($\{X1=nil\}$ at line 4 above). How to read such proofs will be explained in Section 5.

Let us test membership of some ground term. Is the list $\text{cons}(s(s(o)), \text{nil})$ consisting of just the natural number 2 in the intersection? In general, given S as above, a ground term t is recognized at state P if and only if S plus the clause $\perp:-P(t)$ is satisfiable.

```
(cat list_even_inter_tree3plus2.p;\
echo "input_clause(q_rec_cons_2_nil, conjecture,\
  [--q(cons(s(s(o)),nil))]).")\
| h1 -
```

yields

```
*** Derived: q_rec_cons_2_nil ***
```

So $\text{cons}(s(s(o)), \text{nil})$ is in the intersection.

On the other hand, the list $\text{cons}(s(s(s(s(o)))), \text{nil})$ containing just the natural number 4 is *not* in the intersection. Run

```
(cat list_even_inter_tree3plus2.p;\
echo "input_clause(q_rec_cons_4_nil, conjecture,\
    [--q(cons(s(s(s(s(o)))), nil))] )" \
| h1 -
```

and you'll get

(no message at all). Indeed, $\text{cons}(s(s(s(s(o)))), \text{nil})$ is a list of even natural numbers, but not a tree whose numeric leaves are of the form $3n + 2, n \in \mathbb{N}$.

We can do more this way. Is there a term of the form $\text{cons}(s(X), X)$, with the same X , in the intersection? Run

```
(cat list_even_inter_tree3plus2.p;\
echo "input_clause(q_rec_cons_sX_X, conjecture,\
    [--q(cons(s(X), X))] )" \
| h1 -
```

and you'll get

(no message at all). So there is none.

Is there a list whose first element is at least 3 in the intersection? Run

```
(cat list_even_inter_tree3plus2.p;\
echo "input_clause(q_rec_cons_sssX_Y, conjecture,\
    [--q(cons(s(s(s(X))), Y))] )" \
| h1 -
```

and you'll get

```
*** Derived: q_rec_cons_sssX_Y ***
```

So there is one. We don't know which. However, we may use `h1trace` as above to have an idea (left as an exercise!).

Another possibility to have an idea of which lists whose first element is at least 3 in the intersection is to build the automaton recognizing all solutions, by running

```
(cat list_even_inter_tree3plus2.p;\
echo "input_clause(what_q_rec_cons_sssX_Y, conjecture,\
    [++r(X,Y), --q(cons(s(s(s(X))), Y))] )" \
| h1 -no-trim -
mv h1out.model.pl rinter.model.pl
```

The resulting automaton, in file `rinter.model.pl`, is:

```
%[def] __def_1 (Y,X) :- q(cons(s(s(s(X))),Y)).
q(nil).
q(cons(X1,X2)) :- even(X1), tree_3n_plus_2(X1), list_even(X2),
    tree_3n_plus_2(X2).
r(X1,X2) :- two_mod_3(X1), odd(X1), list_even(X2), tree_3n_plus_2
    (X2).
__def_1(X1,X2) :- list_even(X1), tree_3n_plus_2(X1), two_mod_3(X2
    ), odd(X2).
even(o).
even(s(X)) :- odd(X).
list_even(nil).
list_even(cons(X1,X2)) :- even(X1), list_even(X2).
one_mod_3(s(X)) :- zero_mod_3(X).
tree_3n_plus_2(s(X)) :- one_mod_3(X).
tree_3n_plus_2(nil).
tree_3n_plus_2(cons(X1,X2)) :- tree_3n_plus_2(X1), tree_3n_plus_2
    (X2).
two_mod_3(s(X)) :- one_mod_3(X).
odd(s(X)) :- even(X).
zero_mod_3(o).
zero_mod_3(s(X)) :- two_mod_3(X).
```

And graphically, this is the automaton of Figure 4.

3.3.4 Converting Alternating to Non-Deterministic Tree Automata

All right, this starts being a tad intricate. In general, alternating tree automata are not that easy to read. We may eliminate intersection nodes \wedge , and get a *non-deterministic* tree automaton instead by using the `-no-alternation` option to `h1`. Either use

```
h1 -no-trim -no-alternation
```

instead of `h1 -no-trim` (this will produce a non-deterministic, i.e., not an alternating tree automaton, in `rinter.model.pl`), or simply run

```
h1 -no-trim -model
```

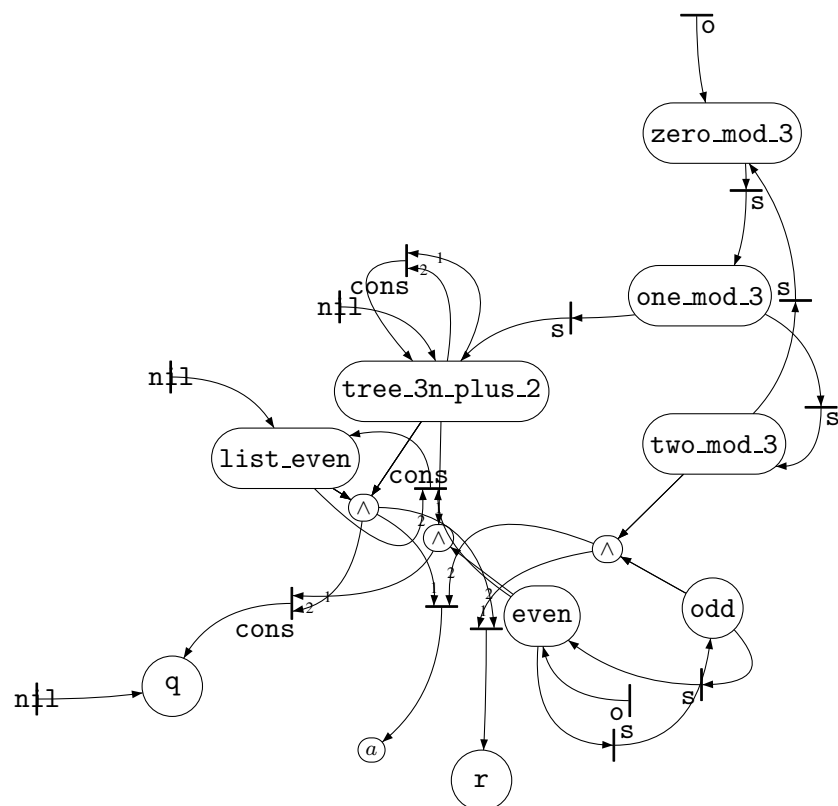
on `rinter.model.pl` to eliminate intersection nodes. Just as

```
h1 -no-trim
```

can be used to convert an \mathcal{H}_1 clause set into an equivalent alternating tree automaton,

```
h1 -no-trim -no-alternation
```

Figure 4: Lists starting with a number at least 3 as recognized in state q of Figure 3



converts an \mathcal{H}_1 clause set, or an alternating tree automaton, into an equivalent non-deterministic tree automaton. Run

```
pl2tptp rinter.model.pl >rinter_nd.p
hl -no-trim -no-alternation rinter_nd.p
```

This yields the following automaton in file `rinter_nd.model.pl`.

```
__inter_odd_and_two_mod_3(s(X)) :- __inter_even_and_one_mod_3
(X).
__inter_odd_and_one_mod_3(s(X)) :-
__inter_even_and_zero_mod_3(X).
q(nil).
q(cons(X1,X2)) :- __inter_even_and_tree_3n_plus_2(X1),
__inter_list_even_and_tree_3n_plus_2(X2).
__inter_even_and_one_mod_3(s(X)) :-
__inter_odd_and_zero_mod_3(X).
r(X1,X2) :- __inter_odd_and_two_mod_3(X1),
__inter_list_even_and_tree_3n_plus_2(X2).
__inter_even_and_tree_3n_plus_2(s(X)) :-
__inter_odd_and_one_mod_3(X).
__inter_list_even_and_tree_3n_plus_2(nil).
__inter_list_even_and_tree_3n_plus_2(cons(X1,X2)) :-
__inter_even_and_tree_3n_plus_2(X1),
__inter_list_even_and_tree_3n_plus_2(X2).
list_even(nil).
list_even(cons(X1,X2)) :- even(X1), list_even(X2).
tree_3n_plus_2(s(X)) :- one_mod_3(X).
tree_3n_plus_2(nil).
tree_3n_plus_2(cons(X1,X2)) :- tree_3n_plus_2(X1), tree_3n_plus_2
(X2).
__inter_even_and_two_mod_3(s(X)) :- __inter_odd_and_one_mod_3
(X).
__inter_odd_and_zero_mod_3(s(X)) :-
__inter_even_and_two_mod_3(X).
two_mod_3(s(X)) :- one_mod_3(X).
one_mod_3(s(X)) :- zero_mod_3(X).
__def_1(X1,X2) :- __inter_list_even_and_tree_3n_plus_2(X1),
__inter_odd_and_two_mod_3(X2).
__inter_even_and_zero_mod_3(o).
__inter_even_and_zero_mod_3(s(X)) :-
__inter_odd_and_two_mod_3(X).
zero_mod_3(o).
zero_mod_3(s(X)) :- two_mod_3(X).
```

```

even(o) .
even(s(X)) :- odd(X) .
odd(s(X)) :- even(X) .

```

Using `pl2gastex` on the output `rinter_nd.model.pl`, we arrive at the non-deterministic tree automaton of Figure 5. This should be more readable. The final state is `q`. Note that the result still contains two copies of the automata recognizing respectively all lists of even numbers, and all trees with leaves of the form *nil* or $3n + 2$, which are not needed any longer.

The automaton of Figure 5, i.e., in file `rinter_nd.model.pl`, uses new states such as `__inter_even_and_one__mod__3` (which recognizes all terms which are both even numbers and numbers of the form $3n + 1$). In general, these new states are named

$$\text{__inter_}P_1P_2\ldots P_n$$

and are meant to recognize all terms that are recognized at P_1 and at P_2 and \ldots and at P_n at the same time. They appear as $P_1 \cap P_2 \cap \ldots \cap P_n$ under `pl2gastex`.

3.3.5 Purging Tree Automata

Well, Figure 5 should be more readable... but there is some junk here. First, there are two sub-automata, disconnected from the rest, defining the predicates `zero_mod_3`, `one_mod_3`, `two_mod_3`, `tree_3n_plus_2`, and `odd`, `even`, `list_even`. They do not contribute at all to the definition of the language of `r`. Second, there are also spurious states such as `q`, or `__aux_1` (drawn as a small state ① or ②). Use `plpurge` to purge the automaton of Figure 5 from all spurious states, by running

```
plpurge -final r rinter_nd.model.pl >rinter_nd.purged.pl
```

Hence we see that the relation `r` is simply the relation relating all numbers that are both odd and equal to 2 modulo 3 (state `odd` \cap `two_mod_3`) to all objects that are both lists of even numbers and trees of with leaves equal to *nil* or $3n + 2$ (state `list_even` \cap `tree_3n_plus_2`).

Looking a bit more in depth, the numbers that are both odd and equal to 2 modulo 3 are 5, 11, 17, \ldots , in other words the numbers that are equal to 5 modulo 6. And the objects that are both lists of even numbers and trees of with leaves equal to *nil* or $3n + 2$ are just lists of numbers equal to 2 modulo 6.

3.3.6 Determinizing Tree Automata

Looking at Figure 6, we realize that taking the successor, i.e., applying the `s` function to a term recognized at `odd` \cap `one_mod_3`, yields a term that is recognized both at `even` \cap `two_mod_3` and at `even` \cap `tree_3n_plus_2`. This is a form of *non-determinism*: we may want to travel to either state, not knowing which will eventually lead to acceptance.

To cater for this, we may *determinize* our tree automata. This produces an equivalent deterministic tree automaton, i.e., a set of Horn clauses of the form

$$P(f(X_1, \ldots, X_n)) \quad :- \quad P_1(X_1), \ldots, P_n(X_n) \tag{5}$$

Figure 5: Eliminating intersection nodes from Figure 4

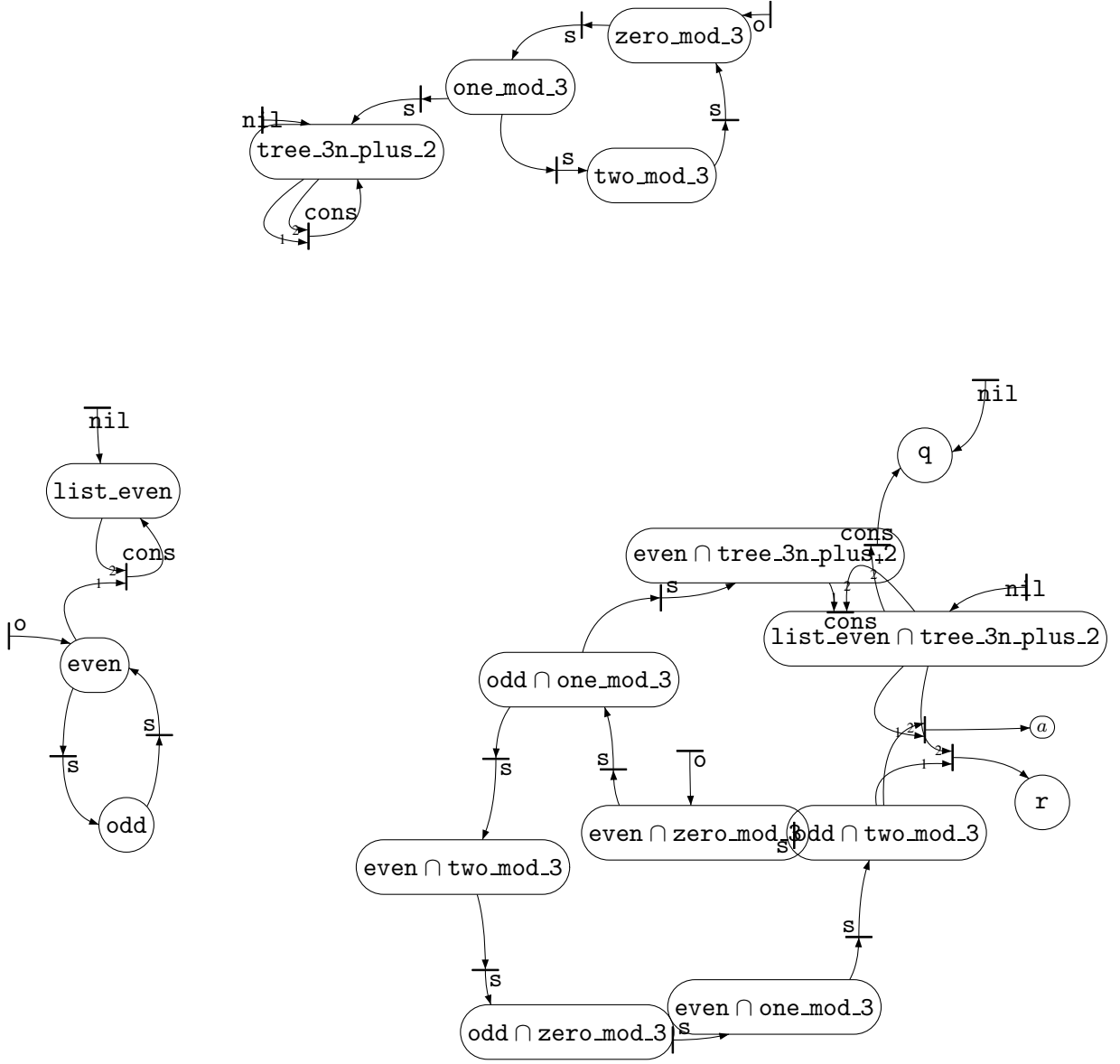
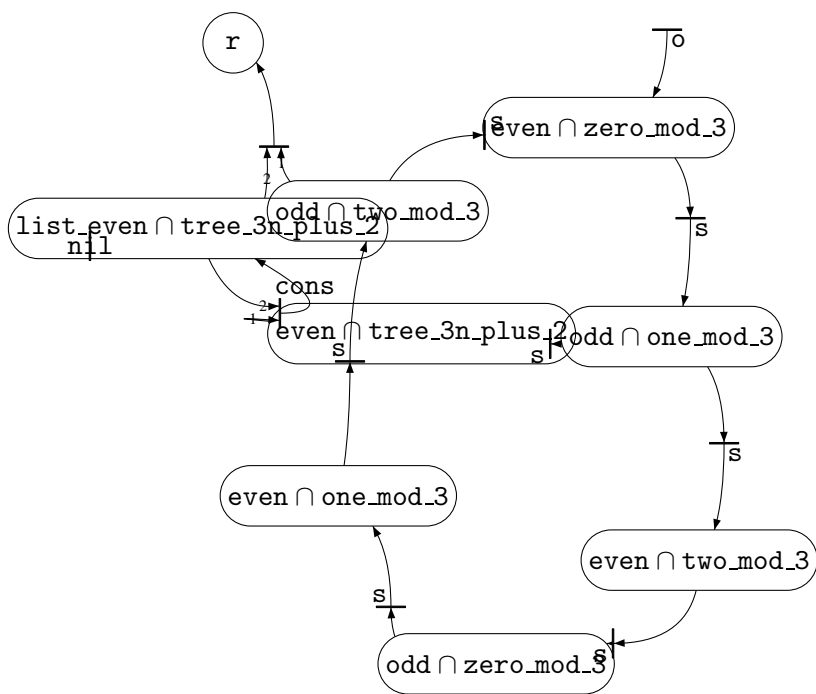


Figure 6: Purging the automaton of Figure 5



where X_1, \dots, X_n are pairwise distinct variables, and where there is *at most* one such clause for each $(n+1)$ -tuple (f, P_1, \dots, P_n) . The automaton of Figure 6 is not deterministic because it contains the two clauses

```

__inter_even_and_tree__3n__plus__2(s(X)) :-
    __inter_odd_and_one__mod__3(X).
__inter_even_and_two__mod__3(s(X)) :- __inter_odd_and_one__mod__3
(X).

```

By definition, a deterministic tree automaton can also be seen as a partial function I_f from tuples of predicates to predicates, one for each f . I.e., $I_f(P_1, \dots, P_n) = P$ if there is a, necessarily unique, clause of the form (5).

To determinize the automaton `rinter.nd.purged.pl` of Figure 6, run

```
pldet rinter_nd.purged.pl >rinter_d.xml
```

This produces a deterministic tree automaton in file `rinter_d.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions>
</definitions>
<states>
  <state name="__exactly_odd_and_two_mod_3">
    <satisfies name="__inter_odd_and_two_mod_3"/>

```

```

</state>
<state name="__exactly_even_and_tree__3n__plus__2">
  <satisfies name="__inter_even_and_two__mod__3"/>
  <satisfies name="__inter_even_and_tree__3n__plus__2"/>
</state>
<state name="__exactly_list__even_and_tree__3n__plus__2">
  <satisfies name="__inter_list__even_and_tree__3n__plus__2"/>
</state>
<state name="__exactly_odd_and_zero__mod__3">
  <satisfies name="__inter_odd_and_zero__mod__3"/>
</state>
<state name="__exactly_odd_and_one__mod__3">
  <satisfies name="__inter_odd_and_one__mod__3"/>
</state>
<state name="__exactly_even_and_one__mod__3">
  <satisfies name="__inter_even_and_one__mod__3"/>
</state>
<state name="__exactly_even_and_zero__mod__3">
  <satisfies name="__inter_even_and_zero__mod__3"/>
</state>
<state name="__bot"/>
</states>
<tables>
  <table name="o" arity="0">
    <entry result="__exactly_even_and_zero__mod__3"></entry>
  </table>
  <table name="s" arity="1">
    <entry result="__exactly_even_and_tree__3n__plus__2"><arg name="__exactly_odd_and_one__mod__3"/></entry>
    <entry result="__exactly_even_and_one__mod__3"><arg name="__exactly_odd_and_zero__mod__3"/></entry>
    <entry result="__exactly_even_and_zero__mod__3"><arg name="__exactly_odd_and_two__mod__3"/></entry>
    <entry result="__exactly_odd_and_two__mod__3"><arg name="__exactly_even_and_one__mod__3"/></entry>
    <entry result="__exactly_odd_and_zero__mod__3"><arg name="__exactly_even_and_tree__3n__plus__2"/></entry>
    <entry result="__exactly_odd_and_one__mod__3"><arg name="__exactly_even_and_zero__mod__3"/></entry>
  </table>
  <table name="nil" arity="0">
    <entry result="__exactly_list__even_and_tree__3n__plus__2"></entry>
  </table>
  <table name="#r" arity="2">
    <entry result="__exactly_r"><arg name="

```

```

        __exactly_odd_and_two__mod__3"/><arg name="
        __exactly_list__even_and_tree__3n__plus__2"/></entry>
</table>
<table name="cons" arity="2">
  <entry result="__exactly_list__even_and_tree__3n__plus__2"><arg
    name="__exactly_even_and_tree__3n__plus__2"/><arg name="
    __exactly_list__even_and_tree__3n__plus__2"/></entry>
</table>
</tables>

```

The format will be explained in more detail in Section 7.

We can then convert this deterministic tree automaton into Prolog notation, since every deterministic tree automaton is a particular case of a non-deterministic tree automaton (itself a particular case of an alternating tree automaton). Use `auto2pl` this way:

```
auto2pl rinter_d.xml >rinter_d.pl
```

This produces a file `rinter_d.pl`, which is probably slightly more readable than the XML file above:

```

__exactly_even_and_zero__mod__3(o).
__exactly_odd_and_two__mod__3(s(X1)) :-
  __exactly_even_and_one__mod__3(X1).
__exactly_odd_and_zero__mod__3(s(X1)) :-
  __exactly_even_and_tree__3n__plus__2(X1).
__exactly_odd_and_one__mod__3(s(X1)) :-
  __exactly_even_and_zero__mod__3(X1).
__exactly_even_and_zero__mod__3(s(X1)) :-
  __exactly_odd_and_two__mod__3(X1).
__exactly_even_and_one__mod__3(s(X1)) :-
  __exactly_odd_and_zero__mod__3(X1).
__exactly_even_and_tree__3n__plus__2(s(X1)) :-
  __exactly_odd_and_one__mod__3(X1).
__exactly_list__even_and_tree__3n__plus__2(nil).
__exactly_list__even_and_tree__3n__plus__2(cons(X1,X2)) :-
  __exactly_even_and_tree__3n__plus__2(X1),
  __exactly_list__even_and_tree__3n__plus__2(X2).
__exactly_r(X1,X2) :- __exactly_odd_and_two__mod__3(X1),
  __exactly_list__even_and_tree__3n__plus__2(X2).

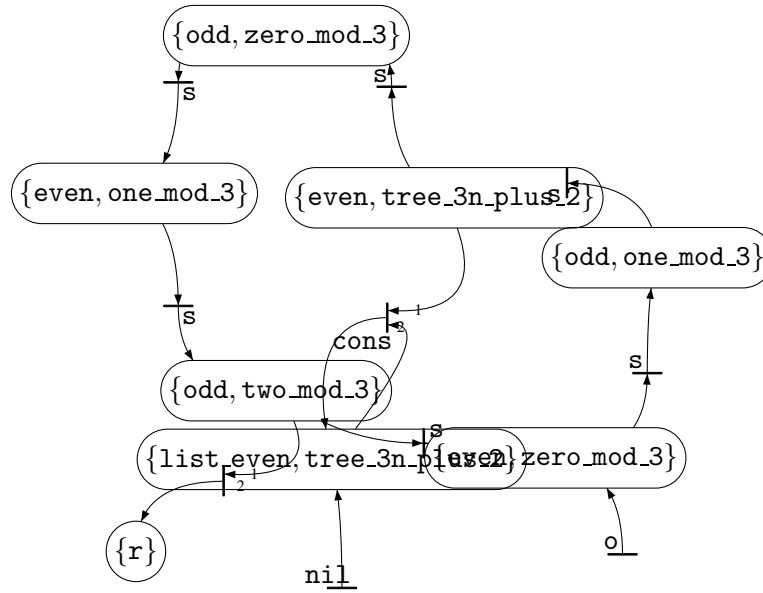
```

Now draw the resulting automaton in Figure 7, using `pl2gastex`:

```
pl2gastex rinter_d.pl >rinterd.tex
```

The automaton of Figure 7 is not too big. But beware: determinizing tree automata may produce automata that are exponentially larger in the general case. In fact, `pldet` may just take forever on some alternating, or even non-deterministic tree automata.

Figure 7: Determinizing the automaton of Figure 6



3.3.7 Computing Unions, Transitive Closures

We have seen how to compute intersections of tree automata in Section 3.3.2. Computing unions is just as easy. Say that you want to compute the union of the sets of lists of even numbers (`listeven.p`) and of the trees whose leaves are `nil` or $3n + 2$, $n \in \mathbb{N}$. That is, instead of computing the intersection of the languages \mathcal{L}_1 and \mathcal{L}_2 introduced at the beginning of Section 3.3, we compute their union. As before, build a file, say `list_even_union_tree3plus2.p`, by concatenating the clauses from `listeven.p` and from `tree3plus2.p`, but this time add the two clauses

```
q(X) :- list_even(X)
q(X) :- tree_3n_plus_2(X)
```

so that the fresh state q recognizes the terms that are recognized by either the state `list_even` or by `tree_3n_plus_2`. Concretely, run the following commands:

```
OUT=list_even_union_tree3plus2
cat listeven.p tree3plus2.p >$OUT.p
echo "input_clause (" $OUT"_1, axiom, \
    [++q(X), --list_even(X)])." >>$OUT.p
echo "input_clause (" $OUT"_2, axiom, \
    [++q(X), --tree_3n_plus_2(X)])." >>$OUT.p
```

Now run `h1 -no-trim` as above:

```
h1 -no-trim list_even_union_tree3plus2.p
```

We get an equivalent alternating tree automaton in the file `list_even_union_tree3plus2.model.pl`:

```
q(nil).
q(s(X)) :- one_mod_3(X).
q(cons(X1,X2)) :- tree_3n_plus_2(X1), tree_3n_plus_2(X2).
q(cons(X1,X2)) :- even(X1), list_even(X2).
two_mod_3(s(X)) :- one_mod_3(X).
odd(s(X)) :- even(X).
zero_mod_3(o).
zero_mod_3(s(X)) :- two_mod_3(X).
one_mod_3(s(X)) :- zero_mod_3(X).
tree_3n_plus_2(nil).
tree_3n_plus_2(s(X)) :- one_mod_3(X).
tree_3n_plus_2(cons(X1,X2)) :- tree_3n_plus_2(X1), tree_3n_plus_2(X2).
list_even(nil).
list_even(cons(X1,X2)) :- even(X1), list_even(X2).
even(o).
even(s(X)) :- odd(X).
```

Graphically, this is the alternating tree automaton of Figure 8, again obtained using `pl2gastex`.

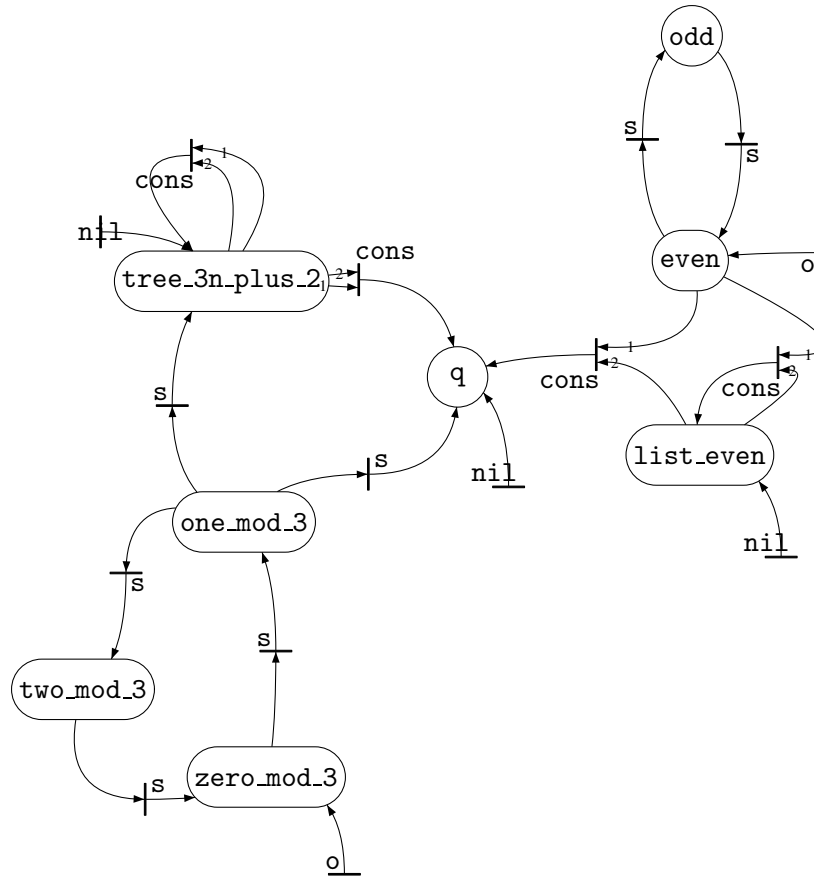
This can be determinized again. Run `pldet` and `auto2pl`:

```
pldet list_even_union_tree3plus2.model.pl \
| auto2pl - >list_even_union_tree3plus2_d.pl
```

and you get

```
__inter_list__even_and_tree__3n__plus__2(nil).
__inter_even_and_zero__mod__3(o).
__inter_even_and_zero__mod__3(s(X1)) :-
    __inter_odd_and_two__mod__3(X1).
__inter_odd_and_zero__mod__3(s(X1)) :-
    __inter_even_and_two__mod__3(X1).
__inter_even_and_one__mod__3(s(X1)) :-
    __inter_odd_and_zero__mod__3(X1).
__inter_odd_and_one__mod__3(s(X1)) :-
    __inter_even_and_zero__mod__3(X1).
__inter_even_and_two__mod__3(s(X1)) :-
    __inter_odd_and_one__mod__3(X1).
__inter_odd_and_two__mod__3(s(X1)) :-
    __inter_even_and_one__mod__3(X1).
```

Figure 8: Trees with leaves equal to nil or to $3n + 2$, or which are lists of even natural numbers



```

tree_3n_plus_2(cons(X1,X2)) :-
    __inter_list__even_and_tree__3n__plus__2(X1),
    __inter_list__even_and_tree__3n__plus__2(X2).
tree_3n_plus_2(cons(X1,X2)) :- tree_3n_plus_2(X1),
    __inter_list__even_and_tree__3n__plus__2(X2).
tree_3n_plus_2(cons(X1,X2)) :-
    __inter_list__even_and_tree__3n__plus__2(X1), tree_3n_plus_2(
        X2).
list_even(cons(X1,X2)) :- __inter_even_and_zero__mod__3(X1),
    __inter_list__even_and_tree__3n__plus__2(X2).
__inter_list__even_and_tree__3n__plus__2(cons(X1,X2)) :-
    __inter_even_and_two__mod__3(X1),
    __inter_list__even_and_tree__3n__plus__2(X2).
tree_3n_plus_2(cons(X1,X2)) :- __inter_odd_and_two__mod__3(X1),
    __inter_odd_and_two__mod__3(X2).
list_even(cons(X1,X2)) :- __inter_even_and_zero__mod__3(X1),
    list_even(X2).
tree_3n_plus_2(cons(X1,X2)) :-
    __inter_list__even_and_tree__3n__plus__2(X1),
    __inter_even_and_two__mod__3(X2).
list_even(cons(X1,X2)) :- __inter_even_and_two__mod__3(X1),
    list_even(X2).
list_even(cons(X1,X2)) :- __inter_even_and_one__mod__3(X1),
    list_even(X2).
tree_3n_plus_2(cons(X1,X2)) :- tree_3n_plus_2(X1), tree_3n_plus_2
    (X2).
tree_3n_plus_2(cons(X1,X2)) :- tree_3n_plus_2(X1),
    __inter_odd_and_two__mod__3(X2).
tree_3n_plus_2(cons(X1,X2)) :- __inter_odd_and_two__mod__3(X1),
    __inter_list__even_and_tree__3n__plus__2(X2).
tree_3n_plus_2(cons(X1,X2)) :- __inter_even_and_two__mod__3(X1),
    __inter_odd_and_two__mod__3(X2).
tree_3n_plus_2(cons(X1,X2)) :- __inter_odd_and_two__mod__3(X1),
    __inter_even_and_two__mod__3(X2).
tree_3n_plus_2(cons(X1,X2)) :-
    __inter_list__even_and_tree__3n__plus__2(X1),
    __inter_odd_and_two__mod__3(X2).
list_even(cons(X1,X2)) :- __inter_even_and_one__mod__3(X1),
    __inter_list__even_and_tree__3n__plus__2(X2).
tree_3n_plus_2(cons(X1,X2)) :- __inter_even_and_two__mod__3(X1),
    tree_3n_plus_2(X2).

```



```

tree_3n_plus_2(cons(X1,X2)) :- __inter_odd_and_two__mod__3(X1),
    tree_3n_plus_2(X2).
tree_3n_plus_2(cons(X1,X2)) :- __inter_even_and_two__mod__3(X1),
    __inter_even_and_two__mod__3(X2).
tree_3n_plus_2(cons(X1,X2)) :- tree_3n_plus_2(X1),
    __inter_even_and_two__mod__3(X2).

```

As you may see, the determinized automaton is much larger this time, and `pl2gastex` now has real trouble trying to draw it. (See Figure 9.) We use `twopi` as graph layout engine here instead of `neato` and run:

```

pl2gastex -v -layout twopi \
    -overlap false \
    list_even_union_tree3plus2_d.pl \
    >list_even_union_tree3plus2_model_d.tex

```

The trick we have used to compute intersections and unions, namely concatenating files and adding new clauses (provided the concatenated files agree on the semantics of predicates), can also be used to compute other combinations of tree languages. A particularly interesting one is the computation of *transitive closures* of relations defined by tree automata (or, more generally, by \mathcal{H}_1 clause sets).

For the purpose of illustration, imagine you want to compute the transitive closure of the binary relation r defined in `rinter.d.pl` (drawn in Figure 7, Section 3.3.6). Just create a fresh binary predicate symbol r^+ , and add the clauses

$$\begin{aligned}
 r^+(X,Y) &:- r(X,Y) \\
 r^+(X,Z) &:- r(X,Y), r(Y,Z)
 \end{aligned}$$

Concretely, run

```

OUT=rinter_d_plus
pl2tptp rinter.d.pl >$OUT.p
echo "input_clause (r_plus_r, axiom, \
    [++r_plus(X,Y), --r(X,Y)])" >>$OUT.p
echo "input_clause (r_plus_tc, axiom, \
    [++r_plus(X,Z), --r_plus(X,Y), --r_plus(Y,Z)])" >>$OUT.p

```

Compute an equivalent non-deterministic tree automata by

```

h1 -no-trim -no-alternation rinter_d_plus.p

```

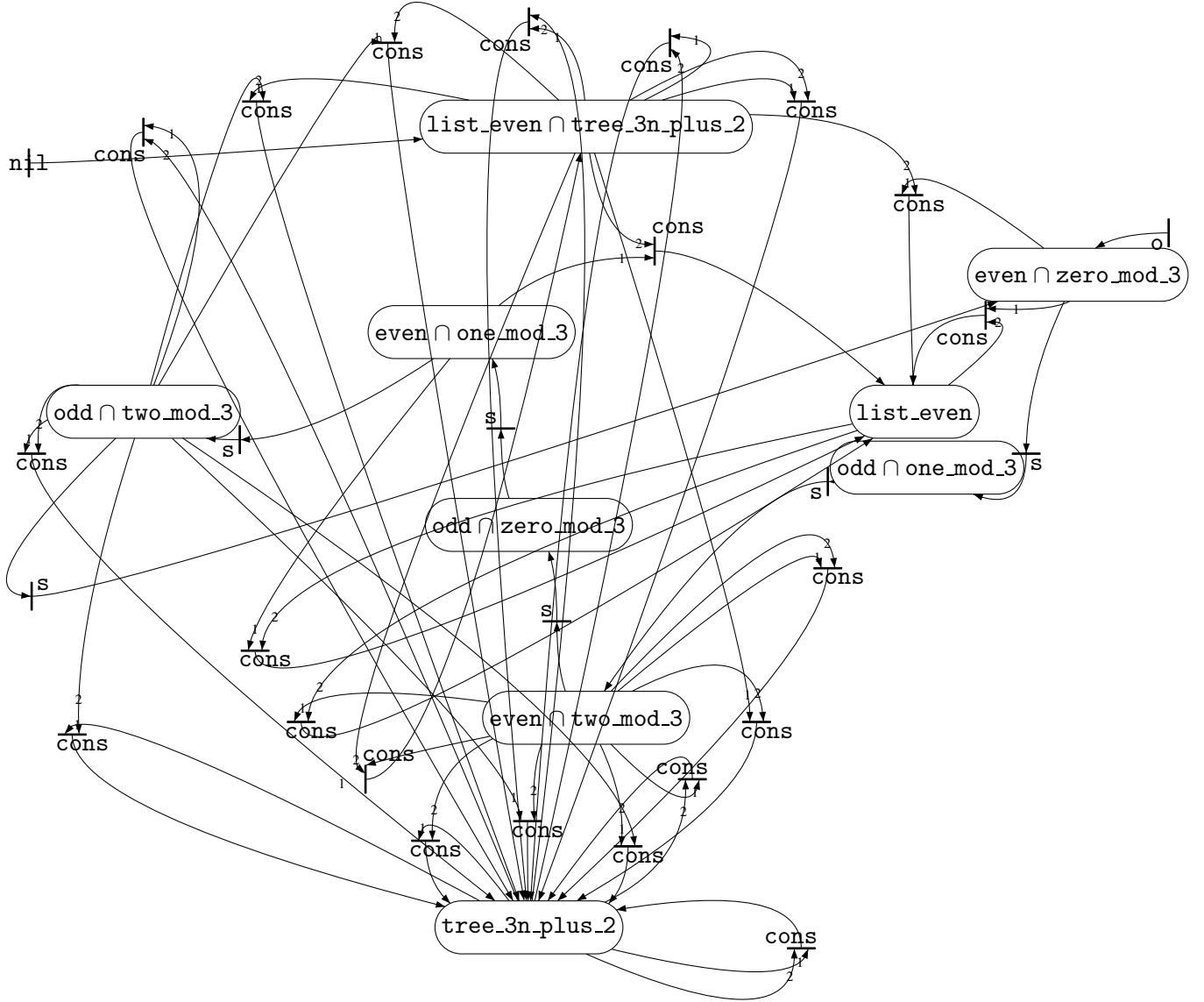
You get

```

__exactly_even_and_tree__3n__plus__2(s(X)) :-
    __exactly_odd_and_one__mod__3(X).
__exactly_even_and_one__mod__3(s(X)) :-
    __exactly_odd_and_zero__mod__3(X).

```

Figure 9: Determinizing the automaton of Figure 8



```

__exactly_odd_and_one__mod__3(s(X)) :-
    __exactly_even_and_zero__mod__3(X) .
__exactly_list__even_and_tree__3n__plus__2(nil) .
__exactly_list__even_and_tree__3n__plus__2(cons(X1,X2)) :-
    __exactly_even_and_tree__3n__plus__2(X1) ,
    __exactly_list__even_and_tree__3n__plus__2(X2) .
__exactly_odd_and_zero__mod__3(s(X)) :-
    __exactly_even_and_tree__3n__plus__2(X) .
__exactly_even_and_zero__mod__3(o) .
__exactly_even_and_zero__mod__3(s(X)) :-
    __exactly_odd_and_two__mod__3(X) .
__exactly_odd_and_two__mod__3(s(X)) :-
    __exactly_even_and_one__mod__3(X) .
__exactly_r(X1,X2) :- __exactly_odd_and_two__mod__3(X1) ,
    __exactly_list__even_and_tree__3n__plus__2(X2) .

```

Since the clauses defining r^+ , i.e., `r_plus`, are the same as those defining `r`, the transitive closure of `r` is r^+ itself here: `r` was already transitive. (Exercise: why?)

3.3.8 Complete Deterministic Tree Automata, Taking Complements

We observed in Section 3.3.6 that a deterministic tree automaton could be seen as a partial function I_f from tuples of predicates to predicates, one for each function symbol f .

A *complete deterministic tree automaton* is a set of Horn clauses of the form (5), i.e.,

$$P(f(X_1, \dots, X_n)) \text{ :- } P_1(X_1), \dots, P_n(X_n)$$

where X_1, \dots, X_n are pairwise distinct variables, and where there is *exactly* one such clause for each $(n+1)$ -tuple (f, P_1, \dots, P_n) (instead of *at most* one such clause for incomplete automata).

Any deterministic tree automaton can be completed to a complete one, by adding a *catch-all* state `_all` (shown as \top by `pl2gastex`), to which all missing transitions are directed. Precisely, if there is an $(n+1)$ -tuple (f, P_1, \dots, P_n) such that there is not clause as above, then add the clause

$$\text{_all}(f(X_1, \dots, X_n)) \text{ :- } P_1(X_1), \dots, P_n(X_n)$$

This must also be done whenever any one of P_1, \dots, P_n is the catch-all state `_all`.

The function I_f is then total. The collection of all such I_f defines a finite model, whose set of values is that of all predicates. A value satisfies a predicate P if and only if it is P , seen as a value.

In fact, finite models and complete deterministic tree automata are exactly the same notion. You might want to ponder this.

As an example, let us return to the Dreadsbury mansion murder mystery (Section 3.2). As we have seen, the only to have possibly killed Aunt Agatha is Aunt Agatha herself. We have proved this by showing that the set of clauses in `butler-puzzle.p` (and explained in Section 3.2) with `WHO` defined as `agatha` was satisfiable.

Since this set is satisfiable, it has a model. Well, `h1` computes such a model, in the guise of an alternating tree automaton. Run

```
cpp -P -DWHO=agatha butler-puzzle.p >agatha.p
hl -no-trim agatha.p
```

and you'll get it in file `agatha.model.pl` (see Figure 10):

```
%[def] __def_4 butler.
%[def] __def_3 X :- not_richer (X,agatha).
%[def] __def_1 agatha.
%[def] __def_5 X :- hates (agatha,X).
%[def] __def_2 charles.
%[def] __def_6 butler.
__def_4(butler).
__def_3(agatha).
__def_1(agatha).
__def_5(charles).
__def_5(agatha).
__def_2(charles).
__def_6(butler).
in_mansion(charles).
in_mansion(butler).
in_mansion(agatha).
hates(X1,X2) :- __def_1(X1), __def_1(X2).
hates(X1,X2) :- __def_1(X1), __def_2(X2).
hates(X1,X2) :- __def_6(X1), __def_5(X2).
hates(X1,X2) :- __def_4(X1), __def_3(X2).
not_richer(X1,X2) :- __def_1(X1), __def_1(X2).
killed(X1,X2) :- __def_1(X1), __def_1(X2).
```

All right, this does not look like a model at all (much less an explanation!), but remember there is a complete deterministic tree automaton that is equivalent to it. We know how to compute an equivalent deterministic tree automaton, using `pldet` and `auto2pl`, viz.

```
pldet agatha.model.pl | auto2pl - >agatha_d.pl
pl2gastex -v agatha_d.pl >agathad.tex
```

Using `pl2gastex`, as usual, produces a visual representation of it, see Figure 11;

So Agatha killed herself, Agatha is the only one not to be richer than Agatha, Agatha hates herself and Charles, the butler hates Agatha and Charles, and Charles hates noone.

Let us now produce the corresponding *complete* deterministic tree automaton. This can be done using `auto2pl`, which we have already seen, using the `-complete 1` option. Be warned, though, that complete deterministic tree automata are large: with n states (included the catch-all states), *any* function taking k arguments will contribute n^k clauses, never less—and do not forget that any k -ary predicate symbol P with $k \geq 2$ creates an invisible function symbol f_P , which will contribute n^k clauses as well. Anyway, run

```
pldet agatha.model.pl | auto2pl -complete 1 - | sort >agatha_c.pl
```

and you'll get the resulting complete deterministic tree automaton in file `agatha_c.pl`.

Figure 10: Why Agatha killed herself

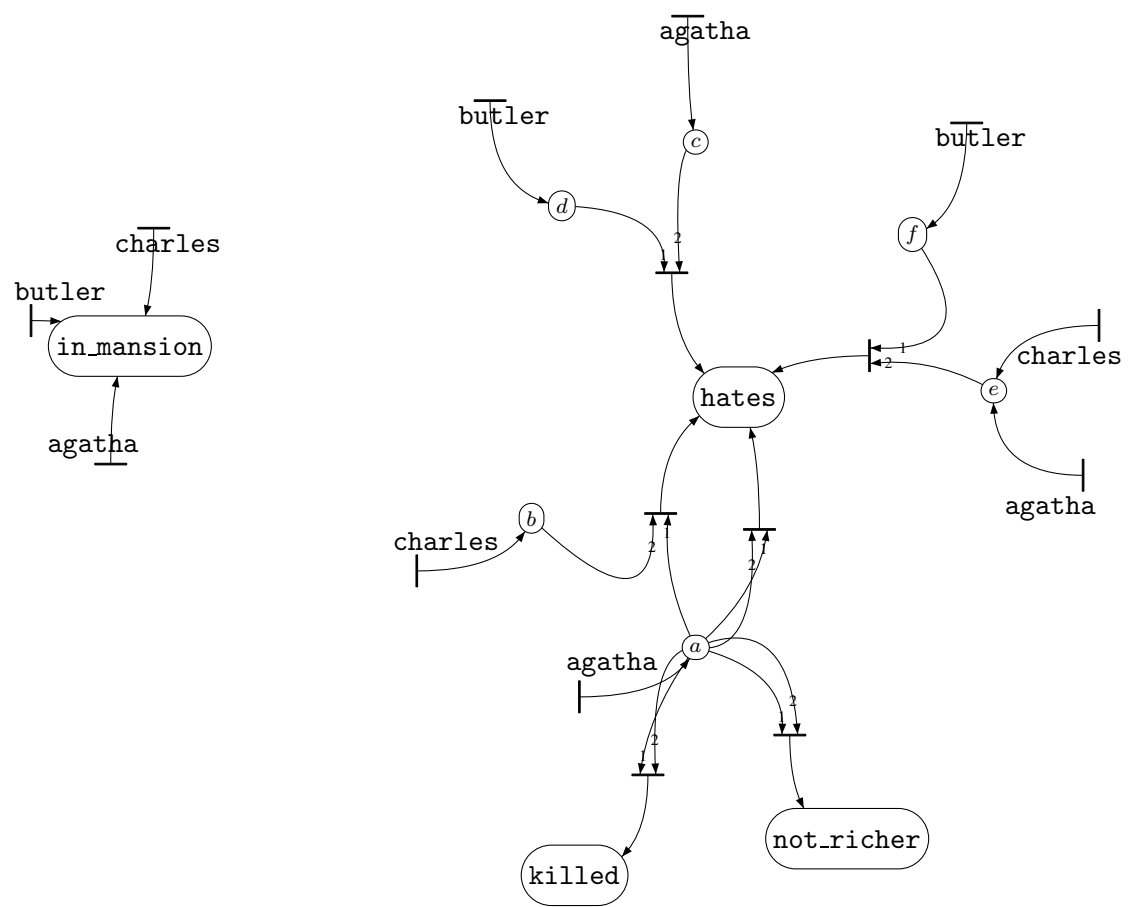
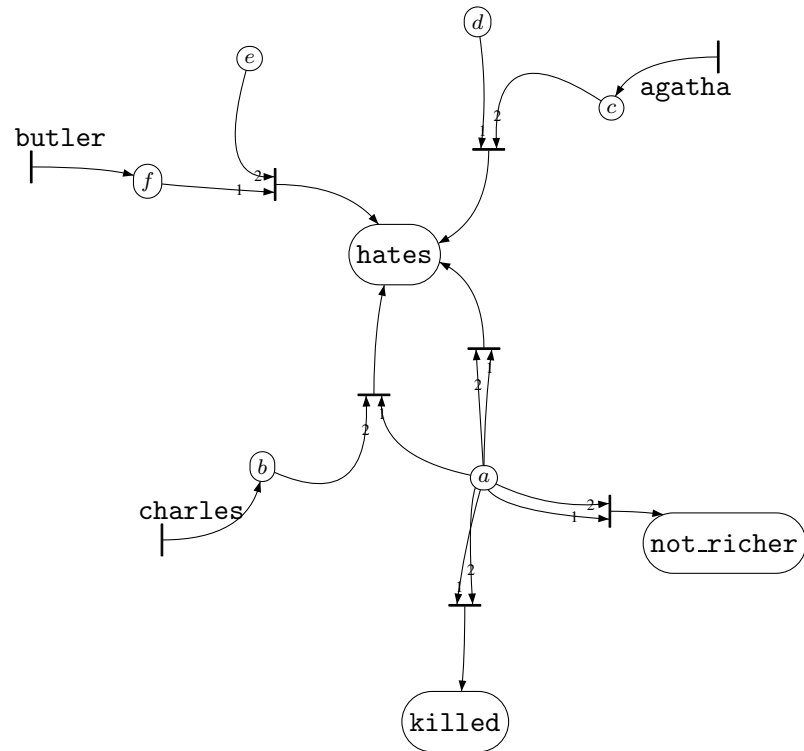


Figure 11: Why Agatha killed herself, deterministically



3.4 The Needham-Schroeder Symmetric Key Protocol Example

Try another example, `nspriv.p`, an encoding of the Needham-Schroeder symmetric key protocol, together with three queries. We do not mean that this is the only possible description of this protocol, and only use this example as a motivation for using `h1` with more complex clause sets.

Here are the clauses of `nspriv.p`, in Prolog notation. We have shown the name of the clause on the left. First, we define a predicate `knows` that is meant to recognize all messages that a malevolent intruder is able to build. The first clauses say that attackers can build any list of messages provided it knows each message in the list, and conversely that it can build any message that appears at any position in a list it knows:

<code>intruder_knows_nil</code>	<code>knows(nil)</code>
<code>intruder_can_take_first_components</code>	<code>knows(M₁) :- knows(cons(M₁, M₂))</code>
<code>intruder_can_take_second_components</code>	<code>knows(M₂) :- knows(cons(M₁, M₂))</code>
<code>intruder_can_build_pairs</code>	<code>knows(cons(M₁, M₂)) :- knows(M₁), knows(M₂)</code>

Lists such as $[M_1, M_2, \dots, M_n]$ are encoded, very classically, as terms $\text{cons}(M_1, \text{cons}(M_2, \dots \text{cons}(M_n, \text{nil}) \dots))$, whence the clauses above. Using a binary symbol *crypt* to denote encryption, i.e., *crypt*(*M*, *K*) denotes the result of encrypting *M* using key *K*, we may also write the following two important rules, stating that the intruder can always encrypt any message it knows using any message it knows, used as a key; and conversely, that the intruder can always decrypt a message if he has the right key.

<code>intruder_can_encrypt</code>	<code>knows(crypt(M, K)) :- knows(M), knows(K)</code>
<code>intruder_can_decrypt_if_has_private_key</code>	<code>knows(M) :- knows(crypt(M, key(pub, K))), knows(key(prv, K))</code>
<code>intruder_can_decrypt_if_has_public_key</code>	<code>knows(M) :- knows(crypt(M, key(prv, K))), knows(key(pub, K))</code>
<code>intruder_can_decrypt_if_has_symmetric_key</code>	<code>knows(M) :- knows(crypt(M, key(sym, X))), knows(key(sym, X))</code>

The last three clauses state how the intruder may decrypt a message of the form *crypt*(*M*, *Z*). We assume that keys come into three varieties, public keys of the form *key*(*pub*, *K*) where *K* is typically the name of the agent holding this public key; private keys of the form *key*(*prv*, *K*) where *K* is the name of the agent holding this private key; and symmetric keys of the form *key*(*sym*, *X*), where *X* is arbitrary. The last three clauses state that you may decrypt a message encrypted with a public key *key*(*pub*, *K*) provided you know the corresponding private key *key*(*prv*, *X*); that you may decrypt a message encrypted with a private key provided you know the corresponding public key; and that you may decrypt a message encrypted with a symmetric key provided you know the latter.

We also assume that there is an operation *s* that builds a new message *s*(*M*) from an old *M*, in bijection with *M*; while we can compute *s*(*M*) from *M* and recover *M* from *s*(*M*), the point is

that M and $s(M)$ always differ.

intruder.can_compute_successors	$\text{knows}(s(M)) \text{ :- knows}(M)$
intruder.can_compute_predecessors	$\text{knows}(M) \text{ :- knows}(s(M))$

In the Needham-Schroeder symmetric key protocol, Alice and Bob communicate with a trusted server to get a common private key that only they know, not the intruder. Alice can always start a session of the protocol and send the server a triple containing Alice's identity *alice*, Bob's identity *bob*, and a nonce, that is, a fresh message for this session. Following Blanchet [2001], we encode this nonce as a function symbol applied to all parameters currently known, say *noncea*(*alice*, *bob*)—the function symbol *noncea* applied to Alice's identity *alice* and Bob's identity *bob*. Now we assume a worst-case intruder model, where any communication can be diverted by the intruder. The net effect is that, from a security viewpoint, what Alice does by sending a message consists exactly in making it known to the intruder:

alice_sends_message_1_to_server	$\text{knows}(\text{cons}(\text{alice},$
	$\text{cons}(\text{bob},$
	$\text{cons}(\text{noncea}(\text{alice}, \text{bob}),$
	$\text{nil}))))$

If the server ever receives such a message, i.e., a message of the form $\text{cons}(A, \text{cons}(B, \text{cons}(Na, \text{nil})))$ for some arbitrary messages A , B , and Na (the server has no way of checking that Alice indeed sent the right message, and can only check the message it receives contains three fields), then it should send out (to Alice, but we have already seen this was irrelevant from a security point of view) the message $\text{crypt}([Na, B, K_{ab}, \text{crypt}([K_{ab}, A], K_{bs})], K_{as})$, where K_{ab} is some fresh key to be used by Alice and Bob, K_{as} is a long-term key shared between Alice and the server, and K_{bs} is a long-term key shared between Bob and the server.

Just like sending a message consists exactly in making it known to the intruder, receiving a message is modeled by stating that the intruder was able to build this message. We shall therefore write a clause saying that, if $\text{knows}[A, B, Na]$ then $\text{knows } \text{crypt}([Na, B, K_{ab}, \text{crypt}([K_{ab}, A], K_{bs})], K_{as})$. Note that we are effectively saying that, from the angle of security, the actions of the server amount to adding new capabilities to the intruder: if the intruder knows a message matching what the server expects, it can build the message that the server will answer, even though it may not know the long-term keys K_{as} and K_{bs} .

Now we encode K_{as} as the term $\text{key}(\text{sym}, \text{cons}(A, \text{cons}(\text{server}, \text{nil})))$, and K_{bs} as the term $\text{key}(\text{sym}, \text{cons}(B, \text{cons}(\text{server}, \text{nil})))$; note that A and B are variables here, representing the fact that the server will find these keys by looking up tables by the identities of A , resp. B . In current sessions, we encode K_{ab} by the term $\text{key}(\text{sym}, \text{current_session}(A, B, Na))$. The key K_{ab} as generated during older sessions is encoded by the term $\text{key}(\text{sym}, \text{old_session}(A, B, Na))$. Separating

current from old sessions means we have to write two clauses:

```

server_answers_A_with_encrypted_packet
  knows(crypt(  cons( Na,
                  cons( B,
                  cons( key(sym, current_session(A, B, Na)),
                  cons( crypt(cons(key(sym, current_session(A, B, Na)),
                                cons(A, nil)),
                                key(sym, cons(B, cons(server, nil)))),
                                nil)))))
                  key( sym, cons(A, cons(server, nil)))))
      :- knows(cons(A, cons(B, cons(Na, nil))))

intruder_knows_previous_server_messages
  knows(crypt(  cons( Na,
                  cons( B,
                  cons( key(sym, old_session(A, B, Na)),
                  cons( crypt(cons(key(sym, old_session(A, B, Na)),
                                cons(A, nil)),
                                key(sym, cons(B, cons(server, nil)))),
                                nil)))))
                  key( sym, cons(A, cons(server, nil)))))
      :- knows(cons(A, cons(B, cons(Na, nil))))

```

All right, now, when Alice receives the message from the server, she should send the part encrypted with K_{bs} to Bob. The idea is that while Alice can decrypt the whole message, which is encrypted with K_{as} , only Bob can decrypt the sub-message that is encrypted with K_{bs} . The message that Alice receives from the server should be $\text{crypt}([Na, B, K_{ab}, \text{crypt}([K_{ab}, A], K_{bs})], K_{as})$, however she can only check that it is of the form $\text{crypt}([Na, B, K_{ab}, Msg])$ for some sub-messages K_{ab} and Msg (which may be totally unrelated to what the server actually sent). She can however check that Na is the nonce $\text{noncea}(\text{alice}, \text{bob})$ that she created earlier (see above, clause `alice_sends_message_1_to_server`), and that B really is Bob's identity `bob`. So Alice expects a message of the form $\text{crypt}([\text{noncea}(\text{alice}, \text{bob}), \text{bob}, K_{ab}, Msg], K_{as})$. As we have said before, for Alice to receive this message, the intruder must send it to Alice, so the intruder must be able to build it. Once Alice receives this, it extract the sub-message Msg and forwards it to Bob—in fact adding it to the set of messages known to the intruder:

```

alice_gets_server_message_and_forwards_submessage_to_bob
  knows(Msg) :- knows(crypt(  cons(noncea(alice, bob),
                                cons(bob,
                                cons(Kab,
                                cons(Msg, nil)))))
                    key(sym, cons(alice, cons(server, nil)))))

```

We use an auxiliary predicate `alice_key` meant to recognize all possible values of K_{ab} above.

This is the key as seen by Alice.

```

alice_gets_server_message_and_stores_current_session_key
  alice_key( $K_{ab}$ )
    :- knows(crypt( cons(noncea(alice, bob),
                        cons(bob,
                        cons( $K_{ab}$ ,
                        cons( $Msg$ , nil))))),
            key(sym, cons(alice, cons(server, nil)))))

```

Let's see what Bob does in this protocol. First, Bob expects to receive the sub-message Msg above. Cutting it short, Bob decrypts it, gets K_{ab} , then sends a confirmation challenge $crypt(N_b, K_{ab})$, where N_b is a fresh nonce. As before, N_b is modeled as a function symbol *nonceb* applied to all relevant variables.

```

agent_B_gets_forwarded_submessage_and_sends_confirmation_challenge
  knows(crypt(nonceb( $K_{ab}$ ,  $A$ ,  $B$ ),  $K_{ab}$ ) :- knows(crypt(cons( $K_{ab}$ , cons( $A$ , nil)),
                                                    key(sym, cons( $B$ , server, nil)))))

```

On receiving this challenge, Alice tries to decrypt it with its own version of the key K_{ab} , and sends back $N_b + 1$:

```

alice_answers_confirmation_challenge
  knows(crypt(s( $N_b$ ),  $K_{ab}$ ) :- alice_key( $K_{ab}$ ), knows(crypt( $N_b$ ,  $K_{ab}$ ))

```

Bob then checks that it indeed gets the confirmation message above with the right value for N_b . If so, we store the key K_{ab} in the predicate *bob_key*:

```

agent_B_checks_confirmation_from_A
  bob_key( $K_{ab}$ ) :- knows(crypt(s(nonceb( $K_{ab}$ ,  $A$ ,  $B$ )),  $K_{ab}$ ))

```

This terminates the description of the protocol. Let us now describe additional things the intruder know. First, the intruder is assumed to know the identities of all agents. We also list who we think are agents. Note that the intruder itself is considered an agent, and has its own identity *intruder*.

```

alice_is_an_agent      agent(alice)
bob_is_an_agent        agent(bob)
server_is_an_agent     agent(server)
intruder_is_an_agent   agent(intruder)
intruder_knows_all_agents  knows( $X$ ) :- agent( $X$ )

```

We also posit that the intruder knows all public keys (... because this is what we mean for them to be public!), and its own private key. We also assume that older sessions are so old that the intruder eventually managed to crack all old sessions key. This is slightly pessimistic. But it is precisely

the purpose of changing session keys to prevent intruders from gaining anything from cracking old session keys.

```

intruder_knows_every_public_key          knows(key(pub, X))
intruder_knows_own_private_key           knows(key(priv, intruder))
intruder_knows_all_previous_session_keys knows(key(sym, old_session(A, B, Na)))

```

This is all, at last.

Let us now ask a few queries. The first asks whether the intruder may know the key K_{ab} as it was generated by the server. The second asks whether the intruder may know any key that Alice accepted as being a key K_{ab} at the end of the protocol. The third asks whether the intruder may know any key that Bob accepted as being K_{ab} at the end of the protocol.

```

intruder_knows_session_key_generated_by_server
  ⊥ :- knows(key(sym, current_session(alice, bob, Na)))
intruder_knows_session_key_as_seen_by_alice
  ⊥ :- alice_key(key(Mode, current_session(X, Y, N))),
        knows(key(Mode, current_session(X, Y, N)))
intruder_knows_session_key_as_seen_by_B
  ⊥ :- knows(crypt(s(nonceb(Kab, A, B)), Kab)), knows(Kab)

```

Now launch h1 on nspriv.p:

```
h1 nspriv.p
```

You should get

```
*** Derived: intruder_knows_session_key_as_seen_by_B ***
```

This means in short that h1 believes that this clause set is unsatisfiable; in terms of protocols, that there is an attack. You cannot actually be sure that this is indeed an attack, i.e., that this clause set is indeed unsatisfiable, because this clause set is not in the \mathcal{H}_1 format. (You should not conclude from this that cryptographic protocols always produce clause sets outside \mathcal{H}_1 , see Nielson et al. [2002].)

In this case, and contrarily to the example of Section 3.2, h1 computes an *approximation* of the input clause set S as a clause set S_1 , and reasons on S_1 instead. Check it using the `-check-h1 2` option:

```
h1 -check-h1 2 nspriv.p
```

and you'll get the list of all clauses in nspriv.p that are not in \mathcal{H}_1 :

```

Warning: clause server_answers_A_with_encrypted_packet has
  non-linear head, variable A occurs repeatedly.
Warning: clause intruder_knows_previous_server_messages has
  non-linear head, variable A occurs repeatedly.
Warning: clause agent_B_gets_forwarded_submessage_and_

```

```

\ sends_confirmation_challenge
has non-linear head, variable Kab occurs repeatedly.
Warning: clause alice_answers_confirmation_challenge has two
non-sibling variables in the head that are connected
in the body, Nb and Kab.
Stop.

```

Note also that `h1` gives you an explanation of what's going wrong. For further explanation of what linear terms, sibling variables and connected variables are, see Nielson et al. [2002] or Goubault-Larrecq [2005].

The `-check-h1 1` option runs the same checks. However, it still proceeds with checking the (un)satisfiability of the approximated clause set S_1 , even when the approximation is not exact, i.e., when the original clause set S is not in \mathcal{H}_1 .

The approximated clause set S_1 is always in the class \mathcal{H}_1 , and always implies S logically. In particular, if `h1` does not find any contradiction, i.e., if S_1 is satisfiable, then S is satisfiable, too. In terms of cryptographic protocols, if `h1` tells you that your protocol has no attack, then you can be sure of it. (That is, up to the accuracy of your model, written as clauses.)

In our case, `h1` found a purported attack, i.e., a purported contradiction. It turns out that `nspriv.p` is indeed contradictory. In fact, in most cases where `h1` thinks a clause set is contradictory, it is indeed contradictory. (Although, as usual, your mileage may vary.)

We terminate this example by mentioning the `-all` option.

The answer

```

*** Derived: intruder_knows_session_key_as_seen_by_B ***

```

above means that `h1` found a (purported) contradiction first, and second that, in order to derive the empty clause \perp , `h1` required the clause `intruder_knows_session_key_as_seen_by_B`. Is this the only query that fails? Remember we have asked three queries. One way of checking this is to remove the `intruder_knows_session_key_as_seen_by_B` clause from `nspriv.p` and run `h1` again. There is a simpler option: run `h1` with the `-all` option. Once a contradiction has been found, instead of stopping just like any other prover, `h1 -all` will continue, trying to find contradictions using other queries.

On the example, running

```
h1 -all nspriv.p
```

you will get the same answer as above:

```

*** Derived: intruder_knows_session_key_as_seen_by_B ***

```

In other words, no other query fails. In terms of security, this means that while Bob's key K_{ab} is not secure, those of Alice and the server definitely are. This may seem paradoxical, however the attack that `h1` just found is one where the intruder deceives Bob into accepted an old, cracked key from an older session. This key is not the key that the server produced. It is also old, meaning that it will be detected as not being new by Alice's use of the nonce N_a .

We have just proved that Alice and the server were in fact safe in this protocol, while Bob is not.

The `-no-resolve` option can be used to disable the reasoning facilities of `h1` altogether. Then `h1 -no-resolve` will just compute an approximation of the input clause set, and store it into the log file. For example,

```
h1 -no-resolve nspriv.p
```

runs without any output, and creates the log file `nspriv.log.gz`. As we have already said, it is not really instructive to look at this file directly with `zcat`. However, you can extract from it the approximated clause set by running:

```
zcat nspriv.log.gz | hlgetlog processed | pl2tptp -
```

This will output the approximated clause set on `stdout`: `hlgetlog` with the `processed` option will extract the approximate clauses and print them in Prolog format. Then `pl2tptp` with `-` argument (meaning `stdin`) will convert these from Prolog format to TPTP format.

All this can be done in a simpler way, by using the `-log-out` option to `h1`. Then `h1` will output its log file, uncompressed, to `stdout`. You can then pipe it to `hlgetlog` and `pl2tptp`, as follows.

```
h1 -log-out -no-resolve nspriv.p | hlgetlog processed | pl2tptp -
```

4 The `h1` Prover

Most of the features of `h1` are explained in Section 3. We explain the structure of the `h1` command line in Section 4.1, then turn to the theoretical basis of `h1` in Section 4.2. We refine this description in Section 4.3, by stating the precise rules used by `h1`.

4.1 How to Use `h1`

The `h1` tool recognizes the following options. As in all Unix tools, options start with a minus sign `-`. Some options are flags, which can be toggled between an active state and a deactivated state. In this case, the convention used in `h1` and other tools of the `h1` suite is to have `-<option>` activate the option, while `-no-<option>` deactivates it.

The options recognized by the `h1` prover are:

- `-h` prints help. This is basically a terse summary of the current explanation.
- `-v<n>` (with no space between `-v` and `<n>`) sets the verbosity level to the integer n . Also, `-v` sets the verbosity level to 1. The default is 0 (run silently).

While other tools of the `h1` suite understand verbosity as a way of printing more or less comments on `stderr`, the `h1` prover itself prints these comments into the log file. That is, into `<filename>.log.gz` if run as `<filename>`, into `h1out.log.gz` if run without a file, i.e., as `h1 -`, and on `stdout` if the `-log-out` option is activated.

This can be useful to examine how `h1` proceeds to derive new clauses and to remove redundant clauses while it is reasoning.

At verbosity level 1, i.e., with `-v1` or `-v`, `h1` adds comments of the form `% | - <clause>` to say which clauses are being derived, and added to the queue of clauses waiting to be processed. Normally, `h1` only lists clauses it is currently processing. At verbosity level 1, `h1` also signals which clauses are useless, in the sense that a simple reachability test on predicate symbols showed that they could not participate in any derivation of the empty clause.

At verbosity level 2, i.e., with `-v2`, `h1` also gives information on its strategy for selecting clauses to be processed, lists the clauses it removes, because they are redundant in some or the other way.

In any case, run `h1 -v1 nspriv.p` or `h1 -v2 nspriv.p`, followed by the command `zcat nspriv.log.gz | less` to see concrete examples of such comments.

The `-v` options are only active if some log file is output at all, that is, if `-log` or `-log-out` is in effect. (By default, `-log` is active.)

- `-check-h1 <n>` enables or disables \mathcal{H}_1 format checking. If $n = 0$, no check is done at all (default). Clauses outside \mathcal{H}_1 are approximated as in Section 3.4, using a slightly more clever variant of the approximation algorithm of Goubault-Larrecq [2005]. If $n = 1$, warnings are printed as to which clauses are not in \mathcal{H}_1 , and what the problem is, then `h1` proceeds approximating all clauses. If $n = 2$, the same warnings are printed; however, if any warning has been emitted, then `h1` will stop and refuse to solve the given clause set.
- `-first`, `-all`: by default, `h1` works as any prover, and stops on finding the first contradiction (`-first`). With the `-all` option, `h1` will report all goal clauses through which a contradiction can be found. See Section 3.4 for an example.

The way `h1` does it is by replacing every goal clause $\perp :- P_1(t_1), \dots, P_n(t_n)$, of name `<name>`, by a definite clause $q_{\langle \text{name} \rangle} :- P_1(t_1), \dots, P_n(t_n)$ for a special nullary predicate symbol $q_{\langle \text{name} \rangle}$, called a *signalling symbol*. With the `-first` option, as soon as `h1` derives a signalling symbol, it prints its name inside `*** Derived:` and `***` and stops. With the `-all` option, `h1` prints the names of all signalling symbols that can be derived.

- `-log`, `-log-out`, `-no-log`: by default, `h1` outputs a trace of what it is doing into a log file (`-log`). If `h1` was called on file `<filename>.p`, then the trace will be in `<filename>.log.gz`. If `h1` was given `-` as input file (i.e., if `h1` reads from standard input), then this file will be `h1out.log.gz`. If `-log-out` is given instead of `-log`, this trace is output, uncompressed, on `stdout`. If `-no-log` is given, then no trace is produced at all.

The reason why the trace is compressed by default is because it may be huge. So huge in fact that I've seen cases where it went above the 2 Gb file size limit on Unix files. To see a rather extreme example, run

```
memtime h1 -progress Fabrice/alice_full1.p
```

This is a 459 clause example, generated automatically from the static analysis by the Csur static analyzer [Goubault-Larrecq and Parrennes, 2005] of a small C implementation of the Needham-Schroeder public-key protocol (*not* the same as the symmetric-key protocol of Section 3.4). Beware that this will take quite some time, and some disk space! On the machine I'm using to write this text (an 1.4 Ghz Pentium IV class machine running RedHat Linux .6.9-1.6_FC2), it takes 33 minutes, and 264 Mb main memory.

Fabrice/alice_full1.p is the largest example we can deal with. Note however that the time and space used by h1 are more related to the structure than to the size of the input problem; e.g., Fabrice/alice_full3.p is about the same size (458 clauses) and apparently very similar, but is shown satisfiable in a matter of seconds by h1.

You may want to run

```
watch h1mon h1.pgr
```

in parallel, to see how h1 progresses on Fabrice/alice_full1.p. (See Section 4.3.)

To cut it short, Fabrice/alice_full1.p will be detected as (possibly) unsatisfiable by h1, and will generate the compressed log file Fabrice/alice_full1.log.gz. The latter is 94.5 Mb. This is not much compared to the uncompressed file, which is a bit more than 40 million lines long, and 2.67 Gb long. These figures are obtained by running

```
zcat Fabrice/alice_full1.log.gz | wc -c -l
```

which itself runs in about 20 s.

- `-model`, `-no-model`: by default, h1 outputs a candidate model of the input clause set, as a set of Horn clauses, into file `<filename>.model.pl`, if h1 was called on file `<filename>.p`, or into `h1out.model.pl` if h1 expects its input from standard input.

The h1 prover is *always* able to output a candidate model, even if the input clause set is unsatisfiable, and therefore has no model. It is just that, in this case, the candidate model is no model: h1mc will then complain, and state which clauses are not satisfied in the candidate model. See Section 6.

To understand what this candidate model may be, especially in the case where there is no model, see Section 4.2: the candidate model is just the subset of all alternating automaton clauses, and universal clauses of the final, saturated clause set that h1 eventually computes.

- `-progress`, `-no-progress`: by default, this is deactivated (`-no-progress`). If activated with `-progress`, a kludge is enabled that allows one to monitor h1's progress: a special file `h1.pgr` is created and written into as h1 goes forward. Each time h1 adds a clause, subsumes a clause, or does anything of the kind, it adds a conventional character to `h1.pgr`. Using `h1mon h1.pgr` at any time while h1 is running will give you instantaneous statistics about how many operations h1 has performed until now. See Section 4.3 for more explanations. Using `watch` in combination with `h1mon`, as in `watch h1mon h1.pgr`, is particularly useful here.

- `-resolve, -no-resolve`. By default (`-resolve`), `h1` computes an \mathcal{H}_1 approximation of the given clause set, then saturates the latter by a form of resolution. The `-no-resolve` disables the resolution engine, so that `h1 -no-resolve` merely produces the \mathcal{H}_1 approximation into the log file (see `-log` option). This is useful to just see what the approximated clauses actually are. To actually see these clauses, run `h1getlog` processed on the log file. See 3.4 for an example.
- `-trim, -no-trim`. By default (`-trim`), `h1` does a quick preprocessing phase (*trimming*) to eliminate some clauses that are obviously not needed to derive a contradiction. This is based on following dependencies between predicate symbols. E.g., if $P(t):-P_1(t_1), \dots, P_n(t_n)$ is an input clause, we say that P depends on P_1, \dots, P_n . A predicate symbol P is needed if and only if there is a goal clause of the form $\perp:-\dots, P(t), \dots$, or if some needed predicate symbol depends on P , inductively. By default, `h1` removes all clauses headed by unneeded predicate symbols.

This is useful to avoid `h1` being clogged by useless clauses. Consider any set of definite clauses (i.e., without any goal clause). While this set of clauses is (obviously!) satisfiable, `h1` may lose quite some time saturating it by resolution. Trimming will just eliminate all clauses, and `h1` will conclude right away that the set of clauses is (obviously, indeed) satisfiable.

This may seem an extreme example. However, this is also an example of why disabling trimming might be the desired option instead. If you use `h1` as a tree automaton tool, as in Section 3.3, you will probably want precisely to feed `h1` with just definite clauses, no goal clause, and expect `h1` to output some compiled alternating tree automaton computing the same languages. To obtain this result, use `-no-trim`.

- `-path-refine <n>`. Another trick that `h1` uses to speed resolution up is to precompute a rather crude over-approximation of the least Herbrand model of the input clause set. See Section 4.2. This consists essentially in computing the sets of paths through atoms in the least Herbrand model, truncated to some fixed length. The `-path-refine` option allows one to specify what the cutoff length should be. By default, it is 3.
- `-alternation, -no-alternation`. By default (`-alternation`), `h1` produces an alternating tree automaton as model in the model file (see `-model` option). With the `-no-alternation` option, `h1` produces a non-deterministic tree automaton. This is in general bigger than the corresponding alternating tree automaton, so that `h1` may take more time saturating clauses with the `-alternation` option. See Section 3.3.4.
- `-deep-abbrev, -no-deep-abbrev`. By default (`-deep-abbrev`), `h1` uses a rule called *abbreviation of deep terms*, which accelerates resolution, sometimes spectacularly. This rule has the following effect. Given some clause $H \Leftarrow P_1(t_1), \dots, P_n(t_n)$ such that, say, t_i is *deep*, i.e., t_i is neither a variable nor a function applied to variables. Then list the free variables of t_i , say X_1, \dots, X_k , create a fresh predicate symbol Q , and replace the above clause by

$$H \Leftarrow P_1(t_1), \dots, P_{i-1}(t_{i-1}), Q(X_1, \dots, X_k), P_{i+1}(t_{i+1}), \dots, P_n(t_n)$$

$$Q(X_1, \dots, X_k) \Leftarrow P_i(t_i)$$

In fact, Q needs not be fresh: if we apply the same rule on the same P_i and the same t_i , we generate the same Q . Using `-no-deep-abbrev` disables this rule. The only use we know of disabling it is measuring how much proof search is sped up by the deep abbreviation rule.

- `-sort-simplify, -no-sort-simplify`. By default (`-sort-simplify`), `h1` uses a rule called *sort simplification*, which originates from the SPASS prover. This is moderately useful in general, but does not seem to be really costly. The idea is as follows. Assume the current set of clauses is S . Amongst these, let S_a be the set of alternating automata clauses. These can be seen as *sort declarations*. E.g., the alternating automaton clause $P(f(X_1, X_2, X_3)) \Leftarrow P_1(X_1), P_2(X_1), P_3(X_2)$ can be read as a rule stating that, if X_1 has sort P_1 and also sort P_2 , if X_2 has sort P_3 , and even if X_3 has no sort, then $f(X_1, X_2, X_3)$ has sort P . Given any clause $H \Leftarrow P_1(t_1), \dots, P_n(t_n)$, extract a *sort environment* ρ : for each i such that t_i is a variable X , ρ states that X has sort P_i . Then, for any j such that t_j is not a variable, if we can deduce that t_j has sort P_j using the sort environment ρ and the current sort declarations, then remove $P_j(t_j)$ from the clause. In principle, this is just a form of resolution, coupled with backward subsumption of the parent clause. However, this variant takes polynomial time, while simulating this by ordinary resolution can take exponential time in the worst case.
- `-monadic-proxy, -standard-approx`. By default (`-standard-approx`), the approximation `h1` uses (see Section 3.4) is a slight refinement of that of Goubault-Larrecq [2005]. This keeps more information, i.e., it is more precise in general, than the probably more intuitive approximation of Frühwirth et al. [1991], which tries to capture the so-called types of variables in each input clause. The `-monadic-proxy` option forces `h1` to use the latter, less precise approximation. The idea is that, by using a less precise approximation, `h1` could run faster. Experience until now has shown, on the contrary, that the less precise approximation is also slower: `h1` generates many spurious clauses that just could not be generated with the more precise approximation.

In other words, `-monadic-proxy` is a false good idea. This option is therefore obsolescent.

- `-body-chop <n>`. Yet another false good idea. Forces `h1` to keep only at most n atoms of depth at least 1 in bodies of clauses generated by resolution. In other words, if `h1` ever tries to generate a clause with $m > n$ atoms of depth at least 1, i.e., of the form $P(f(t_1, \dots, t_k))$, it will savagely remove $m - n$ of them. Resolution then becomes unsound, but will remain complete. That is, if `h1` concludes that the input clause set is satisfiable, it will still be right in saying so. It seems that this in fact does not speed things up at all. This option is therefore obsolescent.

4.2 Theoretical Background

The essentials of the algorithmic underpinnings of `h1` are described, rather tersely, in Goubault-Larrecq [2005]: `h1` implements a form of ordered resolution with selection

4.3 Principle of Operation

5 Explaining and Checking Proofs: `h1trace`, `h1logstrip`

6 Model-Checking Clause Sets and Explaining the Absence of Proofs with `h1mc`

6.1 Theoretical Background

7 Determinizing Tree Automata with `pldet`

8 Converting XML Deterministic Tree Automata to Prolog Notation with `auto2pl`

9 Cleaning and Extracting Automata with `plpurge`

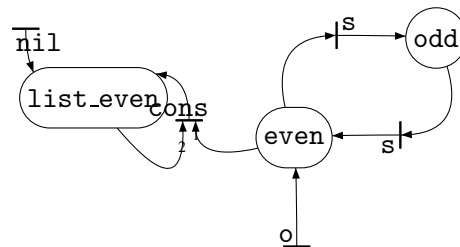
10 Converting Tree Automata and Prolog Programs to TPTP Files

11 Applying Morphisms to \mathcal{H}_1 Clause Sets with `tptpmorph`

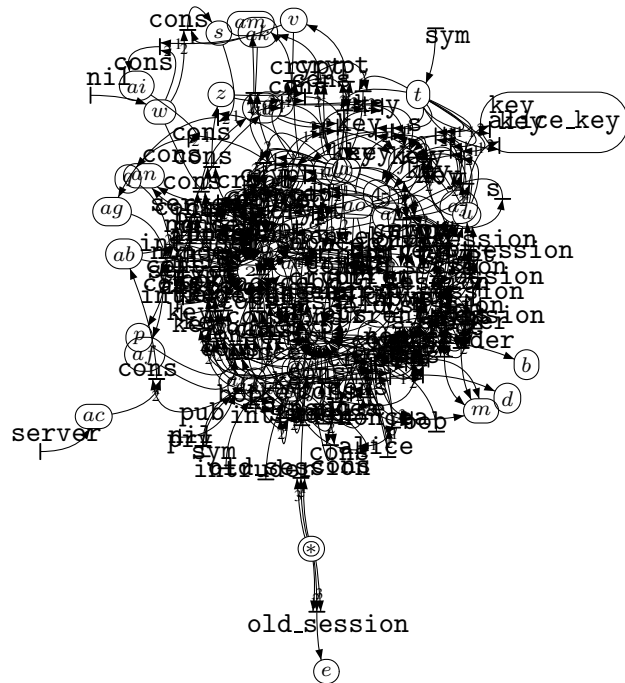
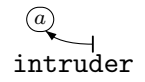
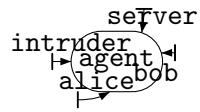
12 Solving Presburger Arithmetic Formulas with `linauto`

13 Displaying Automata with `pl2gastex`

Sometimes, `pl2gastex` gives pretty good results.



On some more complex examples, pl2gastex gives pretty awful results.



14 Log Files and `h1getlog`

15 Bugs

There is no DTD for deterministic tree automata in XML format.

Yes, I know. Sorry. I don't know how to write DTDs. If anybody wants to volunteer, the format should be clear from examples produced by `h1`. Anyone?

Doing `h1 -no-resolve nspriv.p; zcat nspriv.log.gz` fails.

Typically, `zcat` complains of `nspriv.log.gz: unexpected end of file`. This is because `h1` spawns a `gzip -c >nspriv.log.gz` subprocess to compress the log file as it builds it. The `gzip` process can only terminate once `h1` has finished working, but then needs some more time to complete compression. Meanwhile, `zcat` starts decompressing the file, and fails. I currently know of no way to make `h1` close the stream to its subprocess then wait until its subprocess has completed.

If you want to use the log file produced by `h1` in a script file, better use the `-log-out` option, and read it from `stdout`. That is, use `h1 -no-resolve -log-out nspriv.p` instead of `h1 -no-resolve nspriv.p; zcat nspriv.log.gz`.

Clause names are lost in converting clauses to Prolog format.

Yes, this is unfortunate. A few cases have been corrected, by using special comments. But more has to be done.

References

- B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. The Coq proof assistant reference manual: Version 7.4. Rapport technique, INRIA, France, 1999–2003. <http://coq.inria.fr/doc/main.html>.
- B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96. IEEE Computer Society Press, 2001.
- A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In H. Kirchner, editor, *Colloquium on Trees in Algebra and Programming (CAAP'96)*, pages 30–43. Springer Verlag LNCS 1059, 1996.

- P. Devienne, P. Lebègue, A. Parrain, J.-C. Routier, and J. Würtz. Smallest Horn clause programs. *Journal of Logic Programming*, 27(3):227–267, 1996. URL citeseer.nj.nec.com/devienne94smallest.html.
- T. Frühwirth, E. Shapiro, M. Y. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proc. 6th Symp Logic in Computer Science*, pages 300–309, 1991.
- J. Goubault-Larrecq. Deciding \mathcal{H}_1 by resolution. *Information Processing Letters*, 95(3):401–408, Aug. 2005. URL <http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/Goubault-h1.pdf>.
- J. Goubault-Larrecq. Une fois qu’on n’a pas trouvé de preuve, comment le faire comprendre à un assistant de preuve ? In *Actes 15èmes journées francophones sur les langages applicatifs (JFLA’04)*, Sainte-Marie-de-Ré, France, Jan. 2004, pages 1–20. INRIA, collection didactique, 2004. URL <http://www.lsv.ens-cachan.fr/Publis/PAPERS/JGL-JFLA2004.ps>.
- J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In R. Cousot, editor, *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI’05)*, volume 3385 of *Lecture Notes in Computer Science*, Paris, France, Jan. 2005. Springer. URL <http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/GouPar-VMCAI2005.pdf>. To appear.
- F. Nielson, H. R. Nielson, and H. Seidl. Normalizable Horn clauses, strongly recognizable relations and Spi. In *9th Static Analysis Symposium (SAS)*. Springer Verlag LNCS 2477, 2002.
- F. J. Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2:191–216, 1986.
- P. Selinger. Models for an adversary-centric protocol logic. *Electronic Notes in Theoretical Computer Science*, 55(1):73–87, 2001. Proceedings of the 1st Workshop on Logical Aspects of Cryptographic Protocol Verification (LACPV’01), J. Goubault-Larrecq, ed.
- C. B. Suttner and G. Sutcliffe. The TPTP problem library v2.5.0, 2002. URL <http://www.cs.miami.edu/~tptp/TPTP/TR/TPTPTR.shtml>.

Concept Index

Command Index

-all.....	44, 46	sort.....	36
_all.....	35	-no-sort-simplify.....	49
-alternation.....	48	-sort-simplify.....	49
-no-alternation.....	21, 23, 33, 48	-standard-approx.....	49
auto2pl.....	36	T.....	<i>see</i> _all
auto2pl.....	28, 30, 36	-no-trim....15--17, 20, 21, 23,	
-body-chop.....	49	29, 33, 36, 48, 48	
-check-h1.....	14, 43, 44, 46	-trim.....	16, 48
-complete.....	36	twopi.....	33
cpp.....	13, 14	-v.....	45
-deep-abbrev.....	48	watch.....	47
-no-deep-abbrev.....	48	zcat.....	11, 47
-first.....	46		
grep.....	13		
-h.....	45		
h1.pgr.....	47		
hlmon.....	47		
hltrace.....	13, 19		
-log-out.....	19, 45, 46, 46 , 52		
-log.....	46, 46		
-no-log.....	46		
-model.....	21, 47 , 48		
-no-model.....	47		
-monadic-proxy.....	49		
-path-refine.....	48		
pl2gastex.....	16, 24, 28, 35, 36		
pl2tptp.....	23, 33		
pldet.....	26, 30, 36		
plpurge.....	24		
-no-progress.....	47		
-progress.....	47, 47		
-no-resolve.....	45, 48		
-resolve.....	48		