Implementing $\mathcal{H}_1$ by Resolution *

Jean Goubault-Larrecq (goubault@lsv.ens-cachan.fr)
LSV/UMR 8643, CNRS & ENS Cachan; INRIA Futurs projet SECSI

Abstract. The `h1` tool is an implementation of a theorem prover dedicated to solving Nielson, Nielson and Seidl's decidable class $\mathcal{H}_1$ of first-order Horn clauses. This is based on ordered resolution with selection, eager ϵ-splitting—a special case of Riazanov and Voronkov's splitting with naming rule—, and several additional rules. We tested `h1` on a few examples coming from cryptographic protocol verification, and in particular some produced by the `csur` static code analyzer, due to Parrennes and the author, a tool to detect leakage of secrets in C programs. We also tested `h1` on a collection of about 800 problems without equality originating from Sutcliffe and Suttner's TPTP library. We use these examples and report on the efficiency of `h1`. Particularly, we investigate the merits of several optimizations built into `h1`, which appear to be new. In each case, we try to understand why they fare well or fail. These include naive static soft typing, on-the-fly abbreviation of deep terms, and detecting fully-defined predicates. Of the latter three, on-the-fly abbreviation of deep terms, a variant of the rule of definition introduction known in Prolog program transformation circles, offers drastic speedups in specific applications.

Keywords: resolution, splitting, $\mathcal{H}_1$, Horn clauses, abstract interpretation, static soft typing, abbreviation, definition introduction, TPTP, cryptographic protocol verification

* Partially supported by the ACI Rossignol, and the RNTL projects EVA and PROUVÉ.

Keywords: resolution, splitting, $\mathcal{H}_1$, Horn clauses, abstract interpretation, static soft typing, abbreviation, definition introduction, TPTP, cryptographic protocol verification

1. Introduction

Nielson, Nielson and Seidl [16] introduced the class $\mathcal{H}_1$ of first-order Horn clause sets to model reachability in the spi-calculus. They showed that $\mathcal{H}_1$ satisfiability is decidable and DEXPTIME-complete, and that $\mathcal{H}_1$ clause sets can be converted to equivalent tree automata in exponential time—so $\mathcal{H}_1$ defines regular tree languages. Further subclasses of $\mathcal{H}_1$, in particular $\mathcal{H}_3$, have polynomial time complexity.

This was further refined in [9], where it was shown that $\mathcal{H}_1$ could be decided by fairly standard automated deduction techniques, with optimal complexity, that is, in no more than deterministic exponential time. The basic rule is ordered resolution with selection [2], with a suitable ordering and a suitable selection function, supplemented by a form of Riazanov and Voronkov's splitting with naming rule [18].

While this would seem to indicate that off-the-shelf resolution provers could be used to decide $\mathcal{H}_1$, the ones that we tested resisted using the particular selection function that is required here. On the other hand, it is an interesting question in itself whether the particular form of $\mathcal{H}_1$ clauses lends itself to more efficient proof procedures.

This prompted the initial impetus to implement a dedicated resolution prover for the $\mathcal{H}_1$ class. The resulting tool, called `h1`, was implemented and optimized by the author during the years 2003–2004. Some of the lessons learned indeed apply specifically to $\mathcal{H}_1$, but some of the optimizations we found to be most useful are likely to find application in more general situations as well. In this paper, we wish to report on the experience we gained during this effort.

1.1. A word on methodology.

When some implementation effort such as `h1` is initiated by an academic researcher, there is always some confusion as to what the goals of this effort should be. You may want to understand whether some optimization or some deduction rule is useful, and why; or you may strive for, say, the fastest prover you can. Hooker [12] observes that,

while the former is probably the most useful to science, the latter is what most researchers attempt to do.

The second approach is what Hooker calls *competitive testing*: spend some time implementing, optimizing, tuning your code, then compare it against other state-of-the-art provers, e.g., by entering some well-known competition such as the yearly CASC, held at the CADE and IJCAR conferences. Hooker observes that this is essentially useless from the point of view of the advancement of science. You may have won, this may be because of some particularly clever algorithm, or because you are a particularly skillful programmer (what is the part of each?). You may have won because the particular blend of rules and optimizations you used is indeed efficient. In each case, winning a competition does not explain *why* a particular algorithm, rule, or optimization has any benefit, and in which situations.

The first approach is based on *controlled experimentation*, see Hooker (op.cit.). This is the age-old principle at the root of scientific experimentation. The intent is that you run experiments not to boast about your particular algorithm, but to confirm or refute *hypotheses*. E.g., Hooker exemplifies this approach by showing how a controlled experiment could state which of the following hypothesis were likely to explain the efficiency of branching rules in Davis-Putnam-like SAT solvers: a) better branching rules try to maximize the probability that subproblems are satisfiable, and b) better branching rules simplify the subproblems as much as possible. Hooker claims that predictions following from hypothesis a) were consistenly refuted by experiment, so that hypothesis a) had to be abandoned.

We largely agree with Hooker. Nonetheless, we still made the initial mistake of trying to get the most efficient algorithm possible to decide $\mathcal{H}_1$. The `h1` tool was, in our opinion, rather successful from this point of view. During 2003-2004, we concentrated on a few (about 20) problems coming from the application domain we were interested in, cryptographic protocol verification, and specifically, verification that sensitive data remained confidential in C programs [11]. We describe the most prominent examples below. A few weeks before we submitted this paper, we decided that it would be natural to deal with examples coming from the now well-established TPTP collection [20]. More specifically, the $\mathcal{H}_1$ clause sets we picked to test `h1` were:

– Five among the problems coming from cryptographic protocol verification, which were relatively or even extremely challenging at some steps of the development of `h1`. We call them the *Parrennes examples*, since they were communicated to us by Fabrice Parrennes. The examples `alice_full1.h1.p`, `alice_full2.h1.p`, and

`alice_full3.h1.p` that we shall repeatedly refer to later were produced by an early implementation of `csur`, our C code analyzer [11]. The latter files are in fact not in $\mathcal{H}_1$, but they are close. There is a standard approximation procedure (see Section 3.2 below) that converts any first-order clause set S without equality to an $\mathcal{H}_1$ clause set S_0 that over-approximates it, meaning in particular that if S_0 is satisfiable then so is S, and the least Herbrand model of S_0 (which can be presented as a regular tree language) is a model of S.

The files we consider here have gone through this approximation procedure after being generated by `csur`. The early implementation of `csur` we used contained some bugs which, interestingly, generated relatively hard $\mathcal{H}_1$ clause sets. (As the names indicate partly, the examples cited above encode Alice's role in a working C implementation of the Needham-Schroeder public-key protocol [15]. The first too are unsatisfiable—there is an attack—, the third one is satisfiable—no attack.)

The last two examples, which we name `needham_preuve.h1.p` and `needham_saturation.h1.p`, come from the analysis of the same C code, only produced by a later, corrected version of `csur`. Here is an indication of the size of these problems, see Figure 1: we abbreviate, and shall continue to abbreviate, `alice_full1.h1.p` as `a1`, `alice_full2.h1.p` as `a2`, `alice_full3.h1.p` as `a3`, `needham_preuve.h1.p` as `np`, and `needham_saturation.h1.p` as `ns`. Clause size is measured as the number of symbols in it, either constants or variables. But beware that size is essentially independent of the actual difficulty of the problem. (Side note: yes, the names of `needham_preuve.h1.p` ("proof") and `needham_saturation.h1.p` ("saturation") were unfortunately swapped.)

File	# of	#functions				#predicates							avg.	avg.	Sat?
	clauses	per arity			total	per arity						total	#lits/	clause	
		0	1	2		0	1	2	3	4	5		clause	size	
`a1`	1161	349	5	6	360	1	711	3	9	4	2	730	1.63	5.54	No
`a2`	1150	349	5	6	360	0	710	5	9	4	2	730	1.63	5.53	No
`a3`	1153	349	5	5	359	0	715	5	7	5	2	734	1.63	5.55	Yes
`np`	398	98	5	4	107	0	262	7	4	3	2	278	1.71	5.69	Yes
`ns`	397	97	5	4	106	0	260	5	5	3	2	275	1.71	5.71	No

Figure 1. The Parrennes examples

– 808 problems extracted from the TPTP library, version 3.0.1 [20]. Almost none of the problems from the TPTP library are in the $\mathcal{H}_1$ class: only 2 from the COM category (COM001-1.p and COM002-1.p), 1 from KRS (KRS004-1.p), 1 from MGT (MGT036-3.p), 1 from MSC (MSC005-1.p), and 10 from PUZ, totalling 15.

  To compensate, we again used the approximation procedure indicated above. We selected those problems without equality from the COM (computation theory), FLD (fields), KRS (knowledge representation systems), LCL (logic calculi), MGT (management, organization theory), MSC (miscellaneous), NLP (natural language processing), and PUZ (puzzles). We ignored the other categories, which rest heavily on equality. Finding problems without equality was done by executing the Unix command `grep -l ' 0 equa'`, profiting from the fact that each TPTP file comes with extensive descriptions of its properties.

1.2. A Word on Precision

When we build over-approximations of clause sets, a risk is that we lose too much precision. In other words, it might be that the approximated clause set S_0 is unsatisfiable, while S is in fact satisfiable. This didn't happen in the Parrennes examples, which was to be expected since the generated clause sets are already close to $\mathcal{H}_1$ clause sets, i.e., almost all clauses are in $\mathcal{H}_1$. Among the TPTP examples, we sum up our investigations in Figure 2; the "#defin. coarse" row refers to the number of TPTP problems where this loss of precision definitely occurred. This was measured by taking all those sets S_0 judged unsatisfiable by h1 (therefore ignoring the 16 problems that h1 didn't decide, one in COM and 15 in PUZ) and where the corresponding S was judged satisfiable by SPASS v 2.0 [25]. The "#possib. coarse" row counts those additional problems where S_0 was judged unsatisfiable by h1 but we do not know (using SPASS again) whether the corresponding S is satisfiable or not.

	COM	FLD	KRS	LCL	MGT	MSC	NLP	PUZ
#defin. coarse	0	0	6	23	0	1	15	7
#possib. coarse	0	113	0	6	0	0	0	0
Total #	7	279	17	297	22	12	103	71

Figure 2. How often we lose precision via the standard over-approximation

1.3. Competitive Testing

As far as competitive testing is concerned, `h1` does not fare too badly. On the 808 TPTP $\mathcal{H}_1$ clause sets, `h1` solves 792, while SPASS 2.0 solves 795. In both cases, we ran each tool with a 5 minute time limit and a 400 000 Kb memory limit, reflecting reasonable limits on our patience and on the memory capacity of our machine, respectively. These were enforced by using the Unix `memtime` utility with the `-m 400000` and `-c 300` options. All times that we report are user times, not wallclock times, and all memory usage is virtual memory usage.

All problems solved by `h1` are solved by SPASS, too; only 3 problems are solved by SPASS and not by `h1`. (In passing, the main reason why we compare `h1` with SPASS specifically, and not to other provers, is that the sort resolution rule of SPASS decides linear shallow Horn theories specifically [24], and that it turns out that the latter are exactly $\mathcal{H}_1$ clause sets [9].) On the machine on which we tested all implementations, a Pentium IV class machine (1.6 GHz, 512Mb main memory) running Linux 2.6.8-1.521 (Fedora Core 2), statistics are reported in Figure 3 for `h1`, and in Figure 4 for SPASS. "Avg." means average, "St.Dev" stands

Cat.	#problems solved	Time (s.)			Memory (Mb)	
		Total	Avg.	St.Dev.	Avg.	St.Dev.
COM	6	0.790	0.132	0.015	13.0	1.7
FLD	279	46.520	0.167	0.014	14.1	1.4
KRS	17	2.210	0.130	0.008	13.0	1.3
LCL	297	36.890	0.124	0.005	13.2	1.4
MGT	22	2.820	0.128	0.005	12.3	1.2
MSC	12	1.520	0.127	0.006	12.5	1.2
NLP	103	44.500	0.432	1.190	14.5	1.9
PUZ	56	7.390	0.132	0.020	13.2	1.5
Total	792	142.64				

Figure 3. Statistics of `h1` on TPTP Examples

for standard deviation. It should be noted that, certainly because `h1` is implemented in HimML [8], and HimML spends some time at start-up (mainly doing run-time linking of code and data), `h1` always spends a minimum of 0.12 s. per problem, and 12 072 Kb, i.e., 11.79 MB. (This was measured by calling the Unix utility `memtime` on `h1 -h`; the latter starts up, prints a usage banner and exits.)

Cat.	#problems solved	Time (s.) Total	Avg.	St.Dev.	Memory (Mb) Avg.	St.Dev.
COM	7	0.350	0.050	0.079	2.9	0.3
FLD	279	2.650	0.009	0.004	2.7	0.4
KRS	17	0.570	0.034	0.035	2.7	0.7
LCL	297	1.150	0.004	0.047	2.1	7.6
MGT	22	0.870	0.040	0.042	2.9	0.2
MSC	12	0.100	0.008	0.009	2.2	1.1
NLP	103	213.480	2.073	6.197	3.3	0.5
PUZ	58	486.740	8.392	41.922	2.7	2.7
Total	795	705.91				

Figure 4. Statistics of `SPASS` on TPTP Examples

A graphical view of these results is presented in Figure 5 (times in seconds on the x axis, logarithmic scale with $2^{1/4}$ as multiplier between any two consecutive x values; number of problems solved within time x on the y axis) and in Figure 6 (memory in Mb on the x axis).

On the Parrennes examples, the `h1` statistics are given in Figure 9. It is probably the right time to say a word on the precision of these measures. We report means μ and typical variation σ/μ, where σ is standard deviation, over 3 runs on the Parrennes examples, in Figure 10. This indicates that time and memory statistics are to be taken with a grain of salt—typically, times may vary in a $\pm 15\%$ interval around the mean time—while numbers of generated and "automaton" clauses are precise. The "automaton" column counts the number of clauses generated (including those later backward-subsumed) where no literal is selected. These are possibly productive, and describe the least Herbrand model as a tree automaton in the $\mathcal{H}_1$ case (see [16] or [9]) when saturation is reached.

SPASS did not terminate on any of the Parrennes examples, even when allotted 60 minutes of computation time. Remember that the Parrennes examples are approximated problems S_0, stemming from some initial clause set S. SPASS did not terminate on the original, unapproximated problems S either, again within a 60 minute time limit.

We might compare the relative efficiency of `h1` and SPASS (or other provers, for that matter), and propose hypotheses. One of these is that $\mathcal{H}_1$ is a much simpler class to decide than full first-order logic. As we shall see from the rules used in `h1` (Figure 11 below), almost no

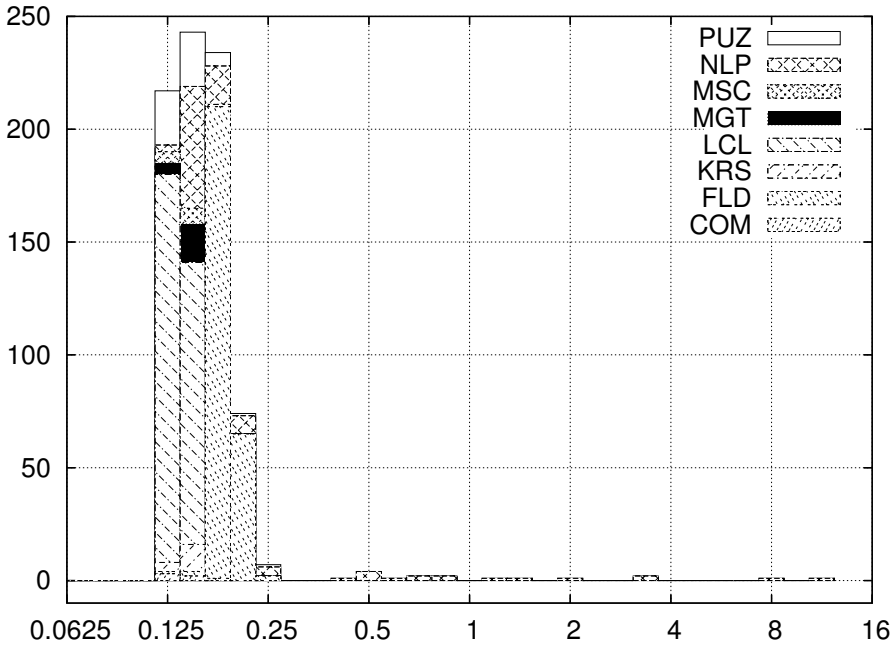

Figure 5. Breakdown of `h1` Times on TPTP Examples

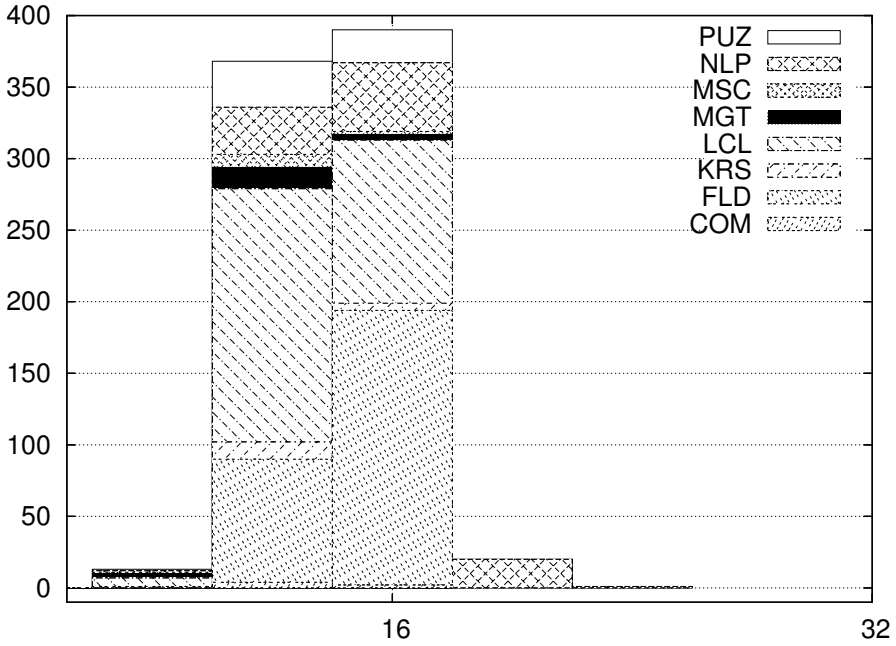

Figure 6. Breakdown of `h1` Memory Usage on TPTP Examples

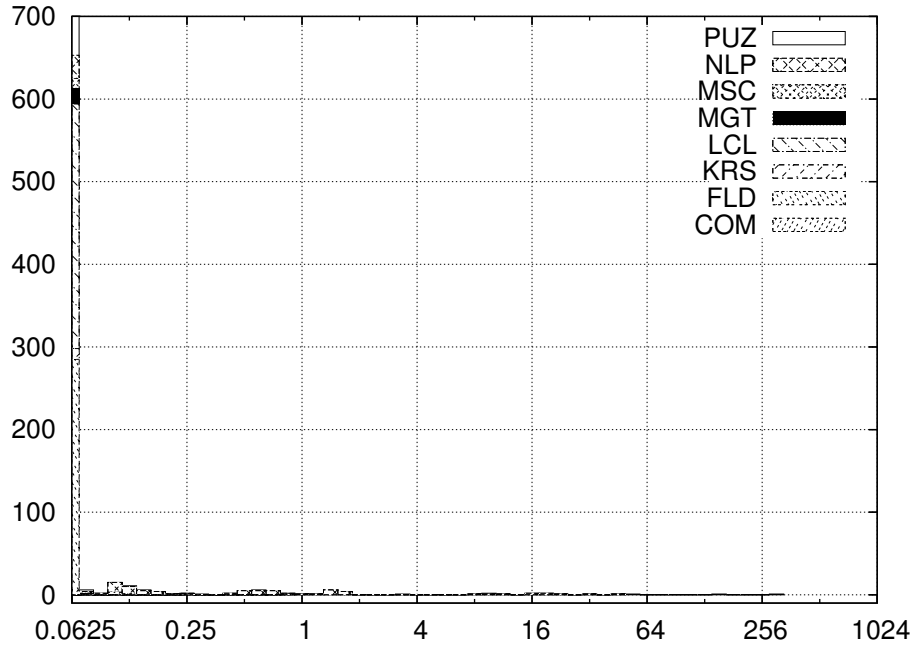

Figure 7. Breakdown of SPASS Times on TPTP Examples

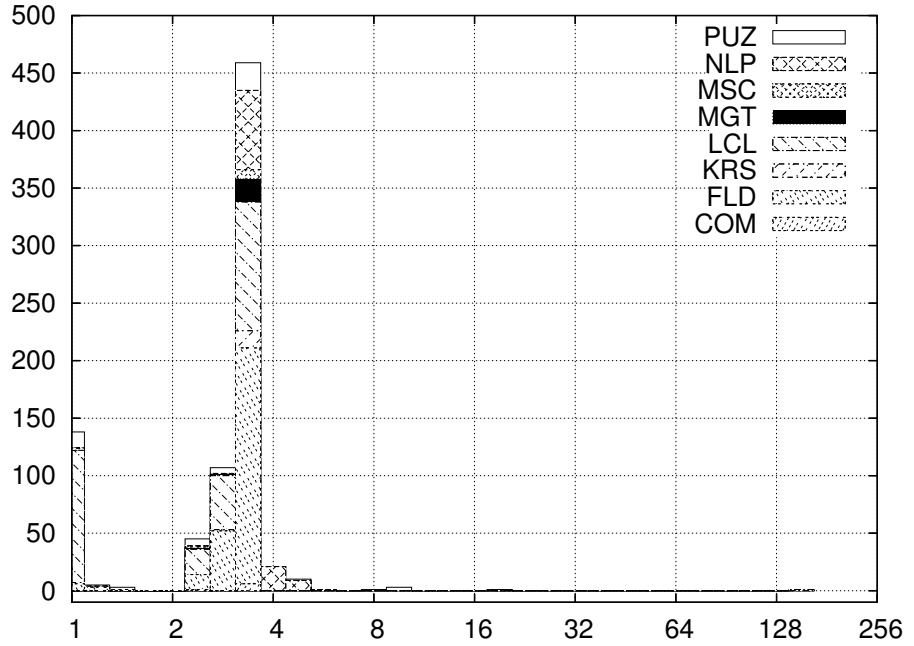

Figure 8. Breakdown of SPASS Memory Usage on TPTP Examples

File	Time (s.)	Memory (Mb)	#clauses generated	#clauses subsumed	#clauses automaton
a1	358.3	262.1	2 912 977	707 816	84 578
a2	273.4	254.3	3 063 812	738 512	85 444
a3	7.6	27.1	83 159	19 223	5 617
np	2.9	19.6	25 062	4 603	2 978
ns	2.2	20.7	28 110	4 665	3 134

Figure 9. Running h1 on the Parrennes Examples

File	Time (s.)	Memory (Mb)	#clauses generated	#clauses subsumed	#clauses automaton
a1	336 ±12%	249 ±3.9%	2.91 10^6 ±0.2%	764 10^3 ±5.2%	84.5 10^3 ±0.1%
a2	352 ±15.8%	275 ±7.0%	3.07 10^6 ±0.2%	796 10^3 ±5.1%	86.1 10^3 ±0.5%
a3	7.3 ±7.5%	29.1 ±6.4%	0.083 10^6 ±0.3%	22.3 10^3 ±9.9%	5.6 10^3 ±0.5%
np	2.8 ±16.2%	20.4 ±3.2%	0.028 10^6 ±6.4%	7.2 10^3 ±25.7%	3.3 10^3 ±6.8%
ns	2.9 ±28.1%	20.3 ±4.6%	0.030 10^6 ±3.7%	7.3 10^3 ±25.7%	3.4 10^3 ±6.1%

Figure 10. Evaluating Variations in Measurements, on the Parrennes Examples

unification is ever needed in deciding $\mathcal{H}_1$. Also, finding and erasing subsumed clauses is probably much easier and faster than in full first-order logic. This may explain that h1 generates roughly about 8 500 clauses per second, while SPASS typically generates about up to 70 per second only, on typical examples. (We should also note that h1 is implemented in HimML, using a prototype HimML bytecode to C compiler, which is currently so bad that the produced C code is roughly only 20 percent faster than the input HimML interpreted bytecode.) One might also argue that h1 has an advantage over competing provers precisely because it is implemented in HimML, where hash-consing [7]

is systematically applied to data of equality-admitting types [8]; in particular, all structures are systematically shared (one may make a parallel with the efficiency of BDDs [3]). This is a hint as to how `h1` in fact just manages to store about 3 million clauses, and all supporting tables and auxiliary data, in just about 300 Mb (see Figure 9 again)—but see the discussion on memory in Section 5.

However, this is not the end of the story. We have actually designed specific rules and techniques to make `h1` run faster. Some of these were tested and found to indeed give some speed or space advantage. This includes our over-approximation scheme (see Section 4) for example. Some others were specifically designed to address specific redundancies. The most notable one is our deep abbreviation scheme (see Section 5). The purpose of this paper is to describe these techniques, examine their efficiency, and attempt to explain why they do work.

Before we end this long introduction, we stress the fact that we do not wish to boast the good performance of `h1` per se. While SPASS 2.0 does not terminate either on the approximated or the original, unapproximated Parrennes examples (at least within one hour), SPASS *does* terminate on the original problems, when run with the `-RSSi=1` (enable sort simplification), `-FuncWeight=999999` (compare terms mainly by function count and ignore predicates), and `-Select=2` (select all negative literals) options. This is how we know for certain the values in the rightmost column of Figure 1. Times are good, too: 45.8 s. for `alice_full1.p`, 62.8 s. for `alice_full2.p`, 0.29 s. for `alice_full3.p`, 0.20 s. for `needham_preuve.p`, 0.27 s. for `needham_saturation.p`. Note that the same options to SPASS are however of no help on the approximated problems: SPASS `-RSSi=1 -FuncWeight=999999 -Select=2` still does not terminate in less than 60 minutes on the (approximated) Parrennes examples. This fact should be taken as a warning that, contrary to commonsense, taking approximations of a problem does not always result in simpler problems. (Although the "#possib. coarse" row in Figure 2 shows that taking approximations indeed resulted in definitely simpler problems on 116 of the 808 TPTP examples.) Our hypothesis that "$\mathcal{H}_1$ is a much simpler class to decide than full first-order logic" above therefore has to be rejected.

1.4. Outline.

We review some related work in Section 2, and give a brief overview of the resolution and splitting techniques at the core of `h1` in Section 3. This is of course needed to understand the rest of the paper, but is not particularly new (see [9]). We also describe more precisely the general clause approximation scheme that we use: this was only outlined in [9,

rules (1), (2)]. We then switch to the core of the paper. Two of the optimizations used in `h1` are particularly effective. The first one computes an over-approximation of the least Herbrand model of the input clause set, and uses it to prune inference steps that stand no chance of being useful. This is described in Section 4. The second consists in introducing fresh predicate symbols to abbreviate deep terms in clause bodies, and is described in Section 5. We consider another optimization, of comparatively lesser importance, in Section 6, full definition subsumption. We conclude in Section 7.

The implementation and provisional documentation of `h1` and its accompanying tools is available from the author's Web page, `http://www.lsv.ens-cachan.fr/~goubault/`.

2. Related Work

2.1. $\mathcal{H}_1$, Linear Shallow Horn Theories

The $\mathcal{H}_1$ class of Nielson, Nielson, and Seidl [16], was used to decide some (over-approximated) reachability properties in the spi-calculus [1]. In [9], the author made the simplifying observation that the $\mathcal{H}_1$ class was equivalent, both in terms of models and with respect to polynomial-time reducibility, to the subclass ${}_\flat\mathcal{H}_1$ ("flat $\mathcal{H}_1$"), a class with a particularly simple definition:

DEFINITION 1. A ${}_\flat\mathcal{H}_1$ *clause* is any Horn clause $H \Leftarrow P_1(t_1), \ldots, P_n(t_n)$, where $t_1, \ldots, t_n$ are arbitrary terms, and the head H is either $\bot$, of the form $P(X)$ or of the form $P(f(X_1, \ldots, X_k))$ where $X_1, \ldots, X_k$ are distinct.

As usual, we fix a first-order signature. Terms are denoted s, t, u, v, ..., predicate symbols P, Q, ..., variables X, Y, Z, ... We assume there are finitely many predicate symbols, and that they are all unary. The latter is fairly innocuous, since we can always replace $P(t_1, \ldots, t_k)$ where $k \geq 1$ with $P(f(t_1, \ldots, t_k))$ for some fresh function symbol f. One consequence of this is that, in Figure 1, `a1` should in fact be considered to contain 730 unary predicate symbols, and 379, not 360, function symbols. Horn clauses C are of the form $H \Leftarrow \mathcal{B}$ where the *head* H is either an atom or $\bot$, and the *body* $\mathcal{B}$ is a finite set $A_1, \ldots, A_n$ of atoms. If $\mathcal{B}$ is empty ($n = 0$), then $C = H$ is a *fact*. If $H = \bot$, then C is a *goal*, otherwise C is a *definite* clause. Note that ${}_\flat\mathcal{H}_1$ clauses are fairly general: only the heads are restricted.

It turns out that Weidenbach had already identified the same class as an interesting, large decidable class of first-order formulas [24]. There,

${}_\flat\mathcal{H}_1$ clause sets are called "monadic Horn theories where all positive literals are linear and shallow". That ${}_\flat\mathcal{H}_1$ can be decided by sort resolution is the topic of [24, Lemma 4]. That ${}_\flat\mathcal{H}_1$ can be decided in deterministic exponential time was shown in [16], and the fact that this optimal upper bound can be achieved with fairly standard resolution techniques was shown in [9].

$\mathcal{H}_1$ and ${}_\flat\mathcal{H}_1$ are *robust* classes, in the sense that essentially any interesting extension of them is undecidable. (The term *robust* used in this empirical sense is attributed to the author by H. Seidl and K. N. Verma. The author thought it was due to K. N. Verma.) The tip of the iceberg is the fact that allowing just one clause with a non-linear head (i.e., with two occurrences of the same variable) produces an undecidable class, see [9, Theorem 11] or [24, end of Section 3]. Unpublished results by the author and by H. Seidl show that various other extensions are undecidable; most of them reduce to the particular encoding of [9, Theorem 11] (mixing $\mathcal{H}_1$ and one-variable clauses; stratified clauses in the style of [13]; etc.)

2.2. Over-approximations, Static Soft Typing

In Section 4 we shall describe a technique to compute an approximation of the least Herbrand model of the initial Horn clause set S. This approximation is used to prune the search space, and can itself be computed rapidly. Using approximations is an old idea, which predates computer science by several centuries. Using an approximation to prune the search space in saturation-based theorem proving is often referred to as *static soft typing* [23].

This preserves completeness. This was proved by C. Weidenbach in the context of ordered resolution with sorts [22, Section 6.7].

2.3. Splitting, Abbreviation, Definition Introduction

We shall use various replacement rules, in addition to resolution. One of them is Riazanov and Voronkov's particular splitting rule [18], which replaces a disjunction $C_1 \vee C_2$ of clauses, where C_1 and C_2 have no variable in common, by the two clauses $C_1 \vee -q$ and $+q \vee C_2$, where q is a fresh predicate constant. (Given any atom A, we write $+A$ for A as a literal, $-A$ for its negation. Any clause is a finite disjunction of literals.) This rule, and in particular the refinement called *splitting with naming*, was shown to give some performance improvement in general first-order theorem proving tasks [18]. We used the latter refinement to give a second proof that $\mathcal{H}_1$ was decidable and DEXPTIME-complete in [9].

The idea of splitting with naming is that q is an abbreviation for the negation of C_2. We replace the disjunction $C_1 \vee C_2$, read as the implication "if C_2 is false, then C_1 is true", by the two implications "if C_2 is false, then q is true" and "if q is true then C_1 is true". (Here we understand C_1 and C_2 as being universally quantified.) One may think of several other, similar, abbreviation schemes. For example, if C_1 and C_2 do share some variables, say X and Y, we may replace $C_1 \vee C_2$ by $C_1 \vee -P(X,Y)$ and $+P(X,Y) \vee C_2$, where P is fresh. The atom $P(X,Y)$ abbreviates the fact that, fixing the values of X and Y, but whatever the values of the other variables in C_2, C_2 is false. This is *non-ground splitting*, a rule that was extensively used in efficient implementations of the MACE finite-model finding procedure [21, 4].

Another related rule is *definition introduction*, used in the field of logic program transformation [17]. This rule replaces every clause $C_1 \vee C_2$, where C_1 and C_2 share no variable, and C_2 is not just a single linear negative literal without function symbols $-P(X_1, \ldots, X_k)$, by the two clauses $C_1 \vee -P(X_1, \ldots, X_k)$ and $+P(X_1, \ldots, X_k) \vee C_2$, where P is a fresh predicate symbol, and $X_1, \ldots, X_k$ are the variables occurring in C_2. As in splitting with naming, we actually reuse the same P whenever we apply definition introduction to clauses with the same C_2 part, up to renaming: $P(X_1, \ldots, X_k)$ abbreviates the fact that C_2 is false for the particular given choice of values for $X_1, \ldots, X_k$. The definition introduction rule was used, together with unfolding (a form of resolution with selection), by Limet and Salzer [14], to obtain decidability results for for inductive definitions over language expressions, and quasi-cs logic programs, which are the largest known decidable classes of set constraints over tree tuple languages.

Our deep abbreviation rule is yet another variation on this theme, see Section 5.

3. Deciding $\mathcal{H}_1$ by Resolution, A Quick Tour

3.1. Ordered Resolution with Selection, ϵ-Splitting

Ordered resolution with selection, with eager ϵ-splitting provides a decision procedure for $\mathcal{H}_1$ [9].

In the case of Horn clauses, ordered resolution with selection can be stated thus: from the *main* premise $A \Leftarrow \mathcal{B}, A_1, \ldots, A_m$ (where $A_1, \ldots, A_m$, $m \geq 1$, is the set of selected atoms if any atom is selected at all, or $m = 1$ and A_1 is $\succ$-maximal in the whole clause if no atom is selected), and the m *side* premises $A'_i \Leftarrow \mathcal{B}'_i$, $1 \leq i \leq m$ (where no atom is selected and A'_i is $\succ$-maximal in each), infer the *resolvent*

$A\sigma \Leftarrow \mathcal{B}\sigma, \mathcal{B}'_1\sigma, \ldots, \mathcal{B}'_m\sigma$ (where σ is the simultaneous most general unifier of A_1 with A'_1, ..., A_m with A'_m).

On the other hand, ϵ-splitting is a particular case of *splitting with naming* [18]. This can be described as follows, refining the intuition of Section 2.3. Call an *ϵ-block* any finite set of atoms of the form $P_1(X), \ldots, P_m(X)$ (with the same X, and $m \geq 0$); it is *non-empty* iff $m \geq 1$. We shall abbreviate ϵ-blocks $B(X)$ to make the variable X explicit. We say that $B(X)$ is a block *of* the clause $A \Leftarrow \mathcal{B}, B(X)$ iff X occurs neither in A nor in $\mathcal{B}$. For each non-empty ϵ-block $B(X)$, create a fresh nullary predicate symbol q_B; if $B(X)$ is a non-empty block of $A \Leftarrow \mathcal{B}, B(X)$, then replace the latter by the two clauses $A \Leftarrow \mathcal{B}, q_B$ and $q_B \Leftarrow B(X)$. Intuitively, the latter *defines* q_B to hold whenever the intersection of the languages of predicates in B is non-empty, and will be called a *defining clause*. The former allows one to conclude A from $\mathcal{B}$, as soon as the *splitting atom* q_B holds.

This replacement rule can be applied anytime without breaking completeness, provided A or $\mathcal{B}$ contains at least one atom of the form $P(t)$ (for any t), and the ordering $\succ$ is extended so that $P(t) \succ q_B$ for every unary predicate symbol P, every term t, and every splitting atom q_B. This is an easy consequence of Bachmair and Ganzinger's standard redundancy criterion [2, Section 4.2.2].

The ordering $\succ$ of [9] is defined by $P(s) \succ Q(t)$ iff s is a proper superterm of t, regardless of P and Q. Define the selection function as follows. Let a *simple* atom be any atom of the form $P(X)$, or $\bot$ by extension. If 1. the clause body contains some splitting atom q_B, select one; otherwise, if 2. it contains a non-simple atom, select one; otherwise, if 3. the head is not simple, then select all atoms in the body; else 4.: none.

It is shown in [9] that these two rules decide the satisfiability of sets of ${}_\flat\mathcal{H}_1$ clauses in exponential time. But we have some freedom in the definition of the selection function. In particular, in case 2., we may choose to select just one non-simple atom, or some of them, or all of them. The `h1` tool selects all non-simple atoms (unless some predicates in the body of the clause are fully defined, see Section 6).

These rules can be presented in the form shown in Figure 11; we explain them below. We write $\overline{X}^k$ for the sequence of pairwise distinct variables X_1, ..., X_k. Note that side premises can only be either *alternating automaton clauses* of the form

$$P(f(X_1, \ldots, X_k)) \Leftarrow B_1(X_1), \ldots, B_k(X_k) \qquad (1)$$

where $B_1(X_1)$, ..., $B_k(X_k)$ are possibly empty ϵ-blocks, or *universal clauses* of the form $P(X)$, or *ne-facts* q_B (stating that B recognizes an *n*on-*e*mpty intersection). This is because side premises have no selected

atom in their bodies, and because we can assume that ϵ-splitting does not apply to them. In the benchmark results presented in the sequel, we shall call "automaton" clauses all alternating automaton and universal clauses alike. These are the ones counted in the "automaton" column of Figure 9 and other similar tables.

Resolution takes different forms, according to the form of the main premise. If the main premise contains a splitting symbol q (case 1. of the selection function), then it must unify with some ne-fact, this is rule (5) of Figure 11. Otherwise, the main premise may be in case 2. of the selection function, where there are $m \geq 1$ selected non-simple atoms $P_1(f_1(t_{11}, \ldots, t_{1k_1})), \ldots, P_m(f_m(t_{m1}, \ldots, t_{mk_m}))$. Some of them, say the first ℓ, will unify with heads of alternating automata clauses, and the others will unify with universal clauses. This is rule (2) when $\ell \geq 1$, and is rule (6) when $\ell = 0$. In case 3. of the selection function, the selected literals must be all (simple) atoms in the body, and the main premise must be of the form $H \Leftarrow P_1(X), \ldots, P_m(X)$ where H is $\bot$, a splitting atom, or an atom $P(X)$ for the same variable X, because of our eager use of ϵ-splitting. In case we unify with $\ell \geq 1$ alternating automaton clauses, either H is $\bot$ or a splitting atom, and we apply rule (3), or $H = P(X)$, and we apply rule (4). If $\ell = 0$, then we again apply rule (6).

One of the things to notice in these rules is the fact that unification and instantiation always takes a very simple form. The role of alternating automaton clauses in rule (2) is to peel off one layer of function symbols $f_1, \ldots, f_m$, and to replace variables X_i by immediate subterms t_{ij} of terms appearing in the selected atoms of the main premise. The other rules are even simpler. A nice byproduct is that no unification procedure is really needed in implementing these rules.

3.2. The Standard Approximation, Converting $\mathcal{H}_1$ to ${}_\flat\mathcal{H}_1$

A nice feature of $\mathcal{H}_1$ and ${}_\flat\mathcal{H}_1$ is that any set of Horn clauses admits a standard approximation, in the form of ${}_\flat\mathcal{H}_1$ clauses. (In fact any set of clauses, Horn or not, by first approximating non-Horn clauses such as $C \vee +A_1 \vee \ldots \vee +A_n$ by the clauses $C \vee +A_1$, $\ldots$, $C \vee +A_n$, as done in Flotter [26].) Recall that we use the letter $\mathcal{B}$ to denote finite sets of atoms—clause *bodies*. In the sequel, $\mathcal{B}\{X := t\}$ denotes $\mathcal{B}$ with t substituted for X. Define the rewrite relation $\rightsquigarrow$ on Horn clause sets by:

$$P(\mathcal{C}[t]) \Leftarrow \mathcal{B} \quad \rightsquigarrow \quad \begin{cases} P(\mathcal{C}[Z]) \Leftarrow \mathcal{B}, Q(Z) & (Z \text{ fresh}) \\ Q(t) \Leftarrow \mathcal{B} \end{cases} \tag{7}$$

where t is not a variable, Q is a fresh predicate symbol,

$$\frac{\overbrace{P_j(f_j(\overline{X}^{k_j})) \Leftarrow B_{j1}(X_1), \ldots, B_{jk_j}(X_{k_j})}^{1\le j\le \ell, \ell\ge 1} \quad \overbrace{P_j(X)}^{\ell+1\le j\le m} \qquad H \Leftarrow P_1(f_1(t_{11}, \ldots, t_{1k_1})), \ldots, P_m(f_m(t_{m1}, \ldots, t_{mk_m})), \mathcal{B}}{H \Leftarrow \mathcal{B}, B_{11}(t_{11}), \ldots, B_{1k_1}(t_{1k_1}), \ldots, B_{\ell 1}(t_{\ell 1}), \ldots, B_{\ell k_\ell}(t_{\ell k_\ell})} \quad (2)$$

$$\frac{\overbrace{P_j(f(\overline{X}^{k})) \Leftarrow B_{j1}(X_1), \ldots, B_{jk}(X_k)}^{1\le j\le \ell, \ell\ge 1} \quad \overbrace{P_j(X)}^{\ell+1\le j\le m} \qquad H \Leftarrow P_1(X), \ldots, P_m(X)}{H \Leftarrow B_{11}(X_1), \ldots, B_{\ell 1}(X_1), \ldots, B_{1k}(X_k), \ldots, B_{\ell k}(X_k)} \quad (3)$$

where $H = \bot$ or H is a splitting literal q

$$\frac{\overbrace{P_j(f(\overline{X}^{k})) \Leftarrow B_{j1}(X_1), \ldots, B_{jk}(X_k)}^{1\le j\le \ell, \ell\ge 1} \quad \overbrace{P_j(X)}^{\ell+1\le j\le m} \qquad P(X) \Leftarrow P_1(X), \ldots, P_m(X)}{P(f(\overline{X}^{k})) \Leftarrow B_{11}(X_1), \ldots, B_{\ell 1}(X_1), \ldots, B_{1k}(X_k), \ldots, B_{\ell k}(X_k)} \quad (4)$$

$$\frac{q \qquad H \Leftarrow \mathcal{B}, q}{H \Leftarrow \mathcal{B}} \quad (5)$$

$$\frac{P_1(X) \ \ldots \ P_m(X) \quad H \Leftarrow \mathcal{B}, P_1(t_1), \ldots, P_m(t_m)}{H \Leftarrow \mathcal{B}} \quad (6)$$

Figure 11. Specializing resolution to ${}_\flat\mathcal{H}_1$ clauses

$$\begin{array}{lcll} & & \text{and } \mathcal{C}[] \text{ is a non-trivial one-hole context} & \\ P(\mathcal{C}[X]) \Leftarrow \mathcal{B} & \rightsquigarrow & P(\mathcal{C}[Y]) \Leftarrow \mathcal{B}, \mathcal{B}\{X := Y\} & (8) \\ \text{where} & & X \text{ occurs at least twice in } \mathcal{C}[X], Y \text{ is a fresh variable} & \end{array}$$

A one-hole context $\mathcal{C}[]$ is a term with a distinguished occurrence of the *hole* $[]$. $\mathcal{C}[u]$ is $\mathcal{C}[]$ with u in place of the hole. $\mathcal{C}[]$ is *non-trivial* iff $\mathcal{C}[] \neq []$. In (8), we require X to occur at least twice, with one occurrence distinguished by the hole in $\mathcal{C}[]$.

The following is shown in [9]. Start from a finite set of Horn clauses S_0. Any $\rightsquigarrow$-normal form of S_0 is an ${}_\flat\mathcal{H}_1$ clause set S_* that logically implies S_0. The relation $\rightsquigarrow$ terminates in polynomially many steps, although not in polynomial time in general. However, it does terminate in polynomial time if we start from clauses having linear heads, such as those already in $\mathcal{H}_1$, as defined by Nielson, Nielson, and Seidl [16].

S_* implies S_0, so, if S_* is satisfiable, then so is S_0, and the least Herbrand model of S_* is a model of S_0. So S_0 is satisfiable, too, and its least Herbrand model is included in that of S_*: it is traditional to call S_* an *over-approximation* (or upper approximation) of S_0.

Using this style of over-approximation is not new, and dates back to [6]. A notable difference, though, is that the above approximation procedure will keep some dependencies between variables which would be thrown away in traditional approximation schemes. This was stressed by [16]. E.g., approximating $P(f(X,Y)) \Leftarrow Q(g(X,f(Y,X)))$—call this clause C—as above will produce the same clause C, thus keeping the relations between X and Y intact; the approximation of [6] would have produced three clauses

$$\begin{aligned} P(f(X,Y)) &\Leftarrow \texttt{type-of-X-in-C}(X), \texttt{type-of-Y-in-C}(Y) \\ \texttt{type-of-X-in-C}(X) &\Leftarrow Q(g(X,f(Y,X))) \\ \texttt{type-of-Y-in-C}(Y) &\Leftarrow Q(g(X,f(Y,X))) \end{aligned}$$

losing the precise relationship between X and Y in the process. The latter style of approximation is achieved by using the `-monadic=2` option in Flotter [26], today called `dfg2dfg`, and we shall call it the *type proxy* approximation in the sequel.

It is easy to show that we can actually refine rules (7) and (8) so as to eliminate spurious dependencies. Given two variables X and Y, and a clause body $\mathcal{B}$, say that X and Y are *connected* in $\mathcal{B}$ [16] if and only if X and Y both occur in the same atom of $\mathcal{B}$, or X and Z are connected and Z and Y are connected for some variable Z, inductively.

We can replace (7) by the following rule

$$P(\mathcal{C}[t]) \Leftarrow \mathcal{B} \quad \rightsquigarrow \quad \begin{cases} P(\mathcal{C}[Z]) \Leftarrow \mathcal{B}_1, Q(Z) & (Z \text{ fresh}) \\ Q(t) \Leftarrow \mathcal{B}_2 \end{cases} \tag{9}$$

obeying the same conditions as (7), and where this time $\mathcal{B}_1$ contains the subset of atoms of $\mathcal{B}$ containing at least one variable connected in $\mathcal{B}$ to some free variable of $\mathcal{C}[Z]$, and $\mathcal{B}_2$ contains at least all other atoms, as well as those containing at least one variable connected to some free variable in t.

Similarly, we can replace (8) by

$$P(\mathcal{C}[X]) \Leftarrow \mathcal{B} \quad \rightsquigarrow \quad P(\mathcal{C}[Y]) \Leftarrow \mathcal{B}_1, \mathcal{B}_2\{X := Y\} \tag{10}$$

with the same side-condition as (8), where $\mathcal{B}_2$ contains all atoms containing X free, and $\mathcal{B}_1$ contains the remaining atoms in $\mathcal{B}$.

The `h1` preprocessor, as well as the `dfg2dfg` tool, implement variants of this; with `dfg2dfg`, use the `-monadic=1`, `-linear`, and `-shallow=3` options.

One can go a bit further. Instead of choosing Q fresh in (9), we may allocate the same Q whenever we come up with the same pair $(t, \mathcal{B}_2)$. Again, this is implemented in the `h1` preprocessor. This benefits

subsequent proof search: if we had generated k fresh symbols for the same pair, then any resolution inference involving $Q(t) \Leftarrow \mathcal{B}_2$ as the main premise would have to be done k times, a clear loss of time. The abbreviation rule of Section 5 is in the same spirit.

In general, one may think that coarser approximations should lead to faster proof search. We have seen in Section 1.3 that this was certainly true for 116 of the 808 TPTP examples, whose unapproximated form is not known to be satisfiable or not, but whose approximated form was (easily) decidable. This was certainly wrong for the Parrennes examples, which are hard, but whose original unapproximated form is reasonably easy (using SPASS with the right options). We should not conclude that there is such a formal connection between precision of an approximation and subsequent efficiency of proof search.

One may also think that coarser approximations will result in extra loss of precision. While this is true, formally, Figure 2 shows that not much precision is lost in general. We do not lose more in general in passing to the type proxy approximation: all of the $\mathcal{H}_1$ clause sets from the Parrennes and TPTP examples whose type proxy approximation was found to be unsatisfiable were indeed unsatisfiable.

4. Quickly Finding Over-approximations of the Least Herbrand Model

Before `h1` starts to saturate a given ${}_\flat\mathcal{H}_1$ clause set using ordered resolution with selection and ϵ-splitting, it computes an over-approximation of the least Herbrand model of the subset of definite clauses.

The first algorithm we implemented to do so was very naive. Strangely enough, and although we tested many variants or even different ideas, the one algorithm that seems to work best is very close to this initial algorithm. Here it is.

4.1. Pathsets

The main data structure is the *pathset*. A pathset is either the special *catch-all* constant Ω, or a finite map (i.e., a table) from pairs (f, i) to pathsets π; the pairs (f, i) consist of a function symbol f, say of arity n, and an integer i, which must be 0 if $n = 0$ (i.e., f is a constant), and otherwise is such that $1 \leq i \leq n$. We require that pathsets are normalized: if $i = 0$, then (f, i) can only be mapped to Ω; moreover, any pair (f, i) is mapped to a *non-empty* pathset π, i.e., either Ω or a non-empty map.

Pathsets are concise representations of sets of paths through ground terms. A *path* is just a finite non-empty sequence of pairs (f, i) as above,

where only the final pair has $i = 0$. Given a ground term t, the *paths through* t are defined as follows. If t is a constant c, then $(c, 0)$ is the only path through t. Otherwise, say $t = f(t_1, \ldots, t_n)$, with $n \geq 1$; then the paths through t are all sequences $(f, i).\pi$, where $1 \leq i \leq n$, and π ranges over the paths through t_i.

In general, the semantics $paths(\psi)$ of pathsets ψ is given as follows: $paths(\Omega)$ is the set of all ground terms, while for any $\psi \neq \Omega$, $paths(\psi) = \{(f, i).\pi | (f, i) \in \operatorname{dom}\psi, i \neq 0, \pi \in terms(\psi(f, i))\} \cup \{(c, 0) | (c, 0) \in \operatorname{dom}\psi\}$.

Computing intersections and unions of pathsets is easy to do, recursively. *Matching* a non-empty pathset ψ against a term t is defined as follows. First, let a *pathset environment* ϱ be any map from variables to non-empty pathsets. Define matching by the rules in Figure 12. The judgment $\varrho \vdash t : \psi \Rightarrow \varrho'$ means that t successfully matches pathset ψ in environment ϱ, producing the refined environment ϱ'. We write $\varrho[X \mapsto \psi]$ the pathset environment mapping X to ψ, and all other variables $Y \in \operatorname{dom}\varrho$ to $\varrho(Y)$.

$$\frac{X \in \operatorname{dom}\varrho \quad \varrho(X) \cap \psi \neq \emptyset}{\varrho \vdash X : \psi \Rightarrow \varrho[X \mapsto \varrho(X) \cap \psi]} \qquad \frac{X \notin \operatorname{dom}\varrho}{\varrho \vdash X : \psi \Rightarrow \varrho[X \mapsto \psi]}$$

$$\frac{}{\varrho \vdash t : \Omega \Rightarrow \varrho} \qquad \frac{\psi \neq \Omega, (c, 0) \in \operatorname{dom}\psi}{\varrho \vdash c : \psi \Rightarrow \varrho}$$

$$\frac{\psi \neq \Omega, (f, 1), \ldots, (f, n) \in \operatorname{dom}\psi \quad \varrho \vdash t_1 : \psi(f, 1) \Rightarrow \varrho_1 \quad \ldots \quad \varrho_{n-1} \vdash t_n : \psi(f, n) \Rightarrow \varrho_n}{\varrho \vdash f(t_1, \ldots, t_n) : \psi \Rightarrow \varrho_n}$$

Figure 12. Matching pathsets

It is easy to see that, for any ground term u such that $t\sigma = u$ for some matching substitution σ, the following holds. Assume that all paths through u are in $paths(\psi)$, then $[] \vdash t : \psi \Rightarrow \varrho'$ is derivable for some pathset environment ϱ'. Moreover, ϱ' is unique and every path through $X\sigma$ is in $paths(\varrho'(X))$, for every $X \in \operatorname{dom}\sigma$. (If $X \notin \operatorname{dom}\varrho'$, we agree that $\varrho'(X)$ denotes Ω.) Moreover, if $[] \vdash t_1 : \psi_1 \Rightarrow \varrho_1$ and $[] \vdash t_2 : \psi_2 \Rightarrow \varrho_2$ are derivable and $\varrho_1(X) \cap \varrho_2(X) \neq \emptyset$ for every variable X, then $\varrho_1 \vdash t_2 : \psi_2 \Rightarrow \varrho_1 \cap \varrho_2$, where $\varrho_1 \cap \varrho_2$ denotes the environment mapping each variable X to $\varrho_1(X) \cap \varrho_2(X)$.

Given an environment ϱ, and any term t, we may compute the pathset $t\varrho$ in the obvious way: if t is a variable X, then $t\varrho = \varrho(X)$; if t is a constant c, then $t\varrho = \{(c, 0) \mapsto \Omega\}$; if t is of the form $f(t_1, \ldots, t_n)$, $n \geq 1$, then $t\varrho = \{(f, i) \mapsto t_i\varrho | 1 \leq i \leq n\}$. The pathset $t\varrho$ is the *instance*

of t by ϱ. Clearly, if every path through $X\sigma$ is in $paths(\varrho(X))$, for every variable $X \in \operatorname{dom}\sigma$, then every path through $t\sigma$ is in $paths(t\varrho)$.

The final operation we need is *chopping*. In general, we require *chop* to be a function from pathsets to pathsets such that $paths(chop(\psi)) \supseteq paths(\psi)$, and such that the range of *chop* is finite. In practice, $chop(\psi)$ replaces any sub-pathset at depth greater than some fixed bound $k \in \mathbb{N}$ by Ω. Formally, $chop = chop_k$, where $chop_0(\psi) = \Omega$, and if $\psi \neq \Omega$ then $chop_{k+1}(\psi) = \{(f,i) \mapsto chop_k(\psi(f,i)) | (f,i) \in \operatorname{dom}\psi\}$.

We then approximate least Herbrand models by *pathset structures*, which are by definition finite maps from predicate symbols to pathsets. Intuitively, if $P(u)$ is in the least Herbrand model of a given set of clauses, we wish to compute a pathset structure Ψ_∞ that will map P to a pathset ψ such that $paths(\psi)$ contains at least all paths through u. Any pathset structure Ψ can also be seen as the set of all pairs (P, π), $P \in \operatorname{dom}\Psi$, $\pi \in paths(\Psi(P))$; implementing unions of pathsets is then formally defined, and again easy to implement.

A clause body $\mathcal{B} = P_1(t_1), \ldots, P_n(t_n)$ *matches* a pathset structure Ψ, yielding the matcher ϱ, if and only if

$$[] \vdash t_1 : \Psi(P_1) \Rightarrow \varrho_1 \quad \ldots \quad \varrho_{n-1} \vdash t_n : \Psi(P_n) \Rightarrow \varrho_n$$

and $\varrho_n = \varrho$. We shall write this $\Psi \Vdash \mathcal{B} \Rightarrow \varrho$. In case no such derivation is possible, we write $\Psi \not\Vdash \mathcal{B} \Rightarrow$.

The desired over-approximation Ψ_∞ can be computed as a least fixpoint of an operator resembling the T_P operator of Prolog semantics. Let S be a set of Horn clauses, and assume for the sake of readability that $\bot$ is just another predicate, so that $\bot\varrho$ is well-defined and equal to $\bot$. For any pathset structure Ψ, define

$$T_S^\sharp(\Psi) \;=\; \Psi \cup \{chop(H\varrho) | \exists (H \Leftarrow \mathcal{B}) \in S \cdot \Psi \Vdash \mathcal{B} \Rightarrow \varrho\}$$

The least fixpoint of $T_S^\sharp$ is the Ψ_∞ we longed for. It is computable, by standard fixpoint iteration algorithms. E.g., let Ψ_0 be the empty pathset, then compute $\Psi_{n+1} = \Psi_n \cup T_S^\sharp(\Psi_n)$ until $\Psi_{n+1} = \Psi_n$. This terminates because *chop* has finite range. Also, it is clear that Ψ_∞ is *sound*, in that: first, if $\bot$ is derivable from S, then $\bot$ is in the domain of Ψ_∞; and second, if the ground atom $P(u)$ is derivable from S, then $paths(\Psi(P))$ contains all paths through u.

In particular, if $\bot$ is not in the domain of Ψ_∞, then S is satisfiable. This very rarely happens in practical situations. The main purpose of Ψ_∞ is different.

Pathsets can then be used to do static soft typing. Typically, if resolution generates a clause $H \Leftarrow \mathcal{B}$, and $\Psi_\infty \not\Vdash \mathcal{B} \Rightarrow$, then remove $H \Leftarrow \mathcal{B}$. Call this rule *naive static soft typing*—naiveté refers to the fact that Ψ_∞ was computed rather naively.

We observe that this preserves completeness in all calculi that can be proved complete by Bachmair and Ganzinger's forcing technique: see [22, Section 6.7]. And ordered resolution with selection, and all forms of splitting we use, are complete by Bachmair and Ganzinger's technique.

In the presence of ϵ-splitting, this involves a subtle point: since the language of predicates is augmented whenever a fresh ne-fact q_B is produced, we need to recompute Ψ_∞. Remember that such ne-facts are generated when ϵ-splitting clauses of the form $A \Leftarrow \mathcal{B}, B(X)$, where X is not free in $A, \mathcal{B}$. To recompute Ψ_∞, we need to adjust Ψ_∞ against the fresh defining clause $q_B \Leftarrow B(X)$ in particular. It is easy to see that doing so merely amounts to adding $\{q_B \mapsto \Omega\}$ to Ψ_∞ whenever the intersection $\bigcap_{P \in B} \Psi_\infty(P)$ is non-empty. When $\bigcap_{P \in B} \Psi_\infty(P)$ is empty, then $\Psi_\infty \not\Vdash B(X) \Rightarrow$, so $\Psi_\infty \not\Vdash \mathcal{B}, B(X)$, and we remove $A \Leftarrow \mathcal{B}, B(X)$ from the current clause set instead.

Call *ne-fact pruning* the special case of naive static soft typing that consists in removing clauses $A \Leftarrow \mathcal{B}, B(X)$ such that $\bigcap_{P \in B} \Psi_\infty(P)$ is empty, and generating q_B only when $\bigcap_{P \in B} \Psi_\infty(P)$ is not empty. The ne-fact pruning rule initially seemed very effective. We have observed, during early stages of the development of `h1`, that ne-fact pruning accounted for a great majority of the cases where naive static soft typing applied. In old versions, naive static soft typing accounted for massive reductions in the search space. In the current version of `h1`, other optimizations muddy the picture.

We have therefore tried to evaluate the impact of naive static soft typing by comparing the results of Section 1.1 with those obtained on the same clause sets, this time with naive static soft typing deactivated. Technically, it suffices to call `h1` with the `-path-refine 0` option, which implements *chop* as $chop_0$, hence computes path structures that map each predicate to Ω. We estimate the overhead of computing with pathsets restricted to Ω (instead of really excising the pathset code) to be negligible.

The results are shown in Figure 13 for the TPTP examples. The number of problems solved is the same. (Previous experiments have exhibited a variation of ± 1 problem solved.) And the total running time was identical, too. However, observe that total running time is a poor indicator of performance, not only because, as we have already seen, we should expect variations of about $\pm 15\%$, but also because running

Cat.	#problems solved	Time (s.)			Memory (Mb)	
		Total	Avg.	St.Dev.	Avg.	St.Dev.
COM	6	0.800	0.133	0.016	13.8	1.5
FLD	279	35.840	0.128	0.004	12.4	1.1
KRS	17	2.230	0.131	0.013	12.4	1.0
LCL	297	37.080	0.125	0.005	12.3	1.0
MGT	22	2.830	0.129	0.005	12.7	1.4
MSC	12	1.530	0.127	0.006	12.7	1.3
NLP	103	52.740	0.512	1.647	13.9	2.1
PUZ	56	7.980	0.142	0.065	13.0	1.6
Total	792	141.03				

Figure 13. Statistics of `h1` on TPTP Examples, without Naive Static Soft Typing

times include both search time *and* the time to compute Ψ_∞. The safest interpretation of Figure 13 is to say that the decrease in efficiency due to not using naive static soft typing must have compensated exactly the decrease in computing time due to our not having to compute Ψ_∞.

It is hard to conclude anything here. In two previous benchmark runs, call them (a) and (b), we observed on the contrary that naive static soft typing resulted in some measurable speedup. In (a), the total times were 42 s. (with naive static soft typing) vs. 147 s. (without). In (b), the corresponding times were 178 s. vs. 271 s. Most of the disparity came from the `NLP` examples, and `PUZ` to a lesser extent, which consistently ran faster with naive static soft typing. The `NLP` problem that took longest in run (a) was `NLP234-1.h1.p`, about 2 minutes. In run (b), this problem took only 1.3 s. In our current experiments, 2.92 s. with naive static soft typing, 0.67 s. without. (Yes, this is faster without.) The `NLP` problem that took longest in run (b) was `NLP237-1.h1.p`, which took 57.4 s. without naive static soft typing, 20.8 with static naive soft typing. In our current experiments, the figures were 5.9 s. (without) and 9.6 s. (with naive static soft typing; yes, this is again faster without). As is apparent, variations in running times are erratic. Variations in numbers of clauses generated are, too, and are in general not even correlated with running times.

The most likely assumption is that the difference between using naive static soft typing and not using it may have had an influence on our clause selection strategy, which classifies clauses according to their weights, for some scale of weights with integer values, and picks clauses

non-deterministically amongst those of equal weights. (We briefly describe our weight system in Section 6.) To be precise, clauses of equal weight are stored in a HimML map [8]. Depending on memory layout, and the history of memory allocation, the order in which elements of a HimML map are stored can vary wildly. Any change in memory allocation may result in a different clause selection strategy. This must have obscured any effect that static soft typing may have had by itself.

On the other hand, most of the TPTP examples are easy. On average, they are solved within 0.178 s., from which, remember, we must discount 0.12 of startup time. To really evaluate naive static soft typing, we should turn to more complex examples, and possibly ones on which clause selection has little effect. The Parrennes examples fit the bill.

Look at Figure 14. Computation times are slower by 0 to 50% without naive static soft typing: 18% for `a1`, 52% for `a2`, 6.5% for `a3`, 0% for `np`, 50% for `ns`. However, we have noticed several times that computation times were a poor indicator of performance. Naive static soft typing does not make much difference in number of clauses generated: only 6% more for the hard examples `a1` and `a2`, 10% more for `a3`, around 20% more for `np` and 40% more for `ns`. (Similarly for number of clauses subsumed on the hard examples; figures are probably less significant on easy examples. In fact, the percentages indicated were consistently the same, up to 1%, on several experiments, except on `ns` where we usually only got 20% more clauses.) The difference is clearer in terms of "automaton" clauses. Remember that "automaton" clauses are those where no literal is selected, and are those that participate in resolution as side premises. Remember also that, although resolution only uses one main premise, it uses as many side premises as there are selected literals in the main premise. So a small increase in "automaton" clauses may result in dramatic loss of performance. Here, naive static soft typing resulted in generating about 30% less "automaton" clauses (`a1`, `a2`), resp. 20% less (`a3`), resp. 40% less (`np`, `ns`). (All percentages given in this paragraph were the same in previous experiments again, except for `np` and `ns`, where the percentage used to be in the region around 20%.)

One figure that does not appear in the above tables is the number of ne-facts generated. With naive static soft typing, they are respectively 658 (1 456 696 clauses split), 718 (1 550 665 clauses split), 173 (36 052 clauses split), 148 (11 713 clauses split), 160 (13 229 clauses split). Without, these numbers become 663 (1 575 574), 715 (1 661 714), 173 (40 834), 151 (16 644), and 162 (18 185). Again, these figures were remarkably impervious to benchmark reruns. We conclude that ne-fact pruning has no measurable impact. Note however that, while only very few ne-facts are generated, the number of clauses split, hence of clauses

File	Time (s.)	Memory (Mb)	#clauses generated	#clauses subsumed	#clauses automaton
a1	421.2	290.7	3 083 535	849 035	110 853
a2	416.7	289.1	3 241 243	874 230	111 604
a3	8.1	31.6	91 022	26 543	6 550
np	2.9	20.7	34 502	10 335	4 183
ns	3.3	20.3	36 176	10 419	4 341

Figure 14. Running h1 on the Parrennes Examples without Naive Static Soft Typing

generated with a splitting symbol q_B in its body, is impressive. One might think of the latter as *frozen* clauses $H \Leftarrow \mathcal{B}, q$, which cannot resolve until rule (5) applies, i.e., until the corresponding ne-fact q is proved. Doing so may then wake up quite a number of clauses $H \Leftarrow \mathcal{B}$, as the numbers above indicate (from 100 to 2 000 on average). However, experience contradicts this opinion: while the number of ne-facts derived varies, no huge variations in the numbers of Figure 14 can be observed.

To conclude this section, we observe that naive static soft typing, using pathsets, offer some moderate benefits, hardly apparent on easy examples, but yielding an improvement in number of "automaton" clauses generated of roughly 30%, and of 20-50% in execution times, on hard examples. More importantly, using naive static soft typing is never detrimental to the efficiency of proof search, at least in any measurable quantity.

4.3. Computing Pathsets Fast

To be effective, the benefits of naive static soft typing should be complemented with a fast algorithm for computing Ψ_∞ initially. This is surprisingly hard to achieve, although the obvious fixpoint algorithm runs in polynomial time.

Our first algorithm was the stupidest possible: let Ψ_0 be the empty pathset, then for $n = 0, 1, \ldots$, compute $\Psi_{n+1} = \Psi_n \cup T_S^\sharp(\Psi_n)$ until $\Psi_{n+1} = \Psi_n$. This naive algorithm terminated in roughly 20 seconds on the a1 example, on an older machine running roughly 3 times slower (the machine we had in 2003, when we evaluated this algorithm).

The current implementation is only moderately less naive. The only difference with the above is that it tries to avoid recomputing facts $chop(H\varrho)$ somewhat. Let S be the input clause set. We first build a table *cindex* mapping each predicate P to the set of clauses in S that

contain an atom of the form $P(t)$ in their body. At each step n of the fixpoint algorithm, we also maintain a set *changed* of predicates P such that $\Psi_{n+1}(P)$ changes compared to $\Psi_n(P)$. Concretely, our algorithm is given in Figure 15, in pseudo-code.

```
fun eval-clause (H ⇐ B) {
    add chop(Hϱ) to Ψ if Ψ ⊩ B ⇒ ϱ;
    if H is of the form P(t) then add P to changed;
}
/* main algorithm */
Ψ := ∅; changed := ∅;
for each C ∈ S do eval-clause (C);
while changed ≠ ∅ {
      S1::=⋃_{P∈changed} cindex(P); changed := ∅;
      for each C ∈ S1 do eval-clause (C);
}
```

Figure 15. Computing the Pathset Structure Ψ_∞

With this implementation, we have not seen yet an instance of a problem where computing Ψ_∞ took more than 0.1 s.

Don't be fooled: we didn't just write the algorithm of Figure 15 and declared ourselves content. Rather, we spent a few months trying to find more clever algorithms, but all of them were desperately inefficient. The most disappointing were worklist algorithms, such as those used at the heart of `h1`'s resolution engine itself. The algorithm of Figure 15 makes a crude attempt at avoiding to recompute already computed expressions $chop(H\varrho)$ by indexing these by their head predicate. Worklist algorithms should be more efficient, because only the really new pathsets have to be considered. So we were surprised to realize that standard worklist algorithms just didn't give any answer after 20 minutes on `a1` (still on a machine roughly 3 times slower than the machine we used in this paper). Looking at traces, we realized that worklist algorithms completely miss the point. Let us illustrate.

Imagine S contains two clauses $P(f(a,b))$ and $P(f(b,a))$, plus a collection of clauses with P in their body. Say there are N of the latter. The worklist algorithm will add both $P(f(a,b))$ and $P(f(b,a))$ to the worklist. Then it will pop one, say $P(f(a,b))$. Given this, it will resolve $P(f(a,b))$ (possibly losing some precision through pathset matching) with all N clauses containing P in their body, pushing up to N new clauses on the worklist. Then it will pop the other atom $P(f(b,a))$, and do the same. Until now, the worklist algorithm took about $2N$ steps.

In general, if there were m unit clauses, not just 2, with head P, the worklist algorithm would take about mN steps until now.

The algorithm of Figure 15, instead, takes $P(f(a,b))$ and $P(f(b,a))$, adds them both to Ψ. This yields a pathset structure mapping P to the four paths $(f,1).(a,0).\Omega$, $(f,2).(b,0).\Omega$, $(f,1).(b,0).\Omega$, and $(f,2).(a,0).\Omega$. (In particular, we lose precision: we cannot rule out matching $P(f(a,a))$ or $P(f(b,b))$ against Ψ.) Next, there are N clauses with P in their bodies. Examining them takes N units of time, for a total of $N+2$—$N+m$ in the general case where there are m, not just 2, initial clauses to consider.

We have therefore replaced mN operations by $m+N$. This holds initially, i.e., at step 0 of the fixpoint algorithm, but in fact at every step of this algorithm. At later steps, the number m of facts to consider in worklist algorithms usually grows as a polynomial in the number of function symbols in the signature (whose degree depends on the level at which terms are chopped), while m is bounded by the number of clauses n_C in the algorithm of Figure 15. Precisely, let n_f be the number of function symbols and n_p the number of predicate symbols, a the maximal arity of function symbols, and k the level at which terms are chopped. Then m can be as large as $n_p(1+n_f(1+n_f(\ldots(1+n_f)^a\ldots)^a) \leq n_p(1+n_f)^{a^k}$ in the worklist algorithm, while $m \leq n_p$ in the algorithm of Figure 15. We have therefore reduced the complexity of finding Ψ_∞ from roughly $n_p(1+n_f)^{a^k}N$ to n_C+N at each step. In the `a1` example, n_f is large but only 11 functions are not constants. The number of possible pathsets is however at least $n_p 11^k N$. With $n_p = 730$, $N \approx n_C = 1\ 161$, and $k=3$, the worklist algorithm may have to consider about 10^9 pathsets at each step while the algorithm of Figure 15 only has to consider at most 2 322… quite a reduction.

4.4. Less Naive Static Soft Typing

Instead of using ad hoc pathset structures, an easy way to compute a more precise over-approximation is to use type proxies. In other words, compute the type proxy approximation S_{proxy} of the given ${}_\flat\mathcal{H}_1$ clause set S, then run `h1` itself a first time on S_{proxy}. Hopefully, this should run faster than on S directly. If S_{proxy} is satisfiable, then S is, too, and we are done. Otherwise, `h1` will output an alternating automaton (i.e., sets of clauses of the form (1), plus universal clauses) that describes a finite model $\mathcal{M}$ of the subset of definite clauses in S_{proxy}. (To get the model explicitly, determinize and complete the automaton; the states of the resulting automaton $\mathcal{A}$ form the domain of the model, and the value of f on values $q_1, \ldots, q_n$ is the unique state q such that there is a transition $q(f(X_1,\ldots,X_n)) \Leftarrow q_1(X_1),\ldots,q_n(X_n)$ in $\mathcal{A}$. More details

in [10].) Using $\mathcal{M}$, a collection of regular tree languages, as an over-approximation of the least Herbrand model of S if any, is closer to the original spirit of static soft typing for resolution.

However, computing $\mathcal{M}$ is slow. One can guess so already, considering that $\mathcal{M}$ will be computed by using a worklist algorithm. In fact, computing $\mathcal{M}$ is roughly as slow, or as fast, as deciding the ${}_\flat\mathcal{H}_1$ clause set S directly: the results on the TPTP examples are given in Figure 16, which we invite the reader to compare with those of Figure 3.

Cat.	#problems solved	Time (s.) Total	Avg.	St.Dev.	Memory (Mb) Avg.	St.Dev.
COM	6	0.810	0.135	0.008	12.5	1.3
FLD	279	46.250	0.166	0.015	13.6	1.5
KRS	17	2.240	0.132	0.012	12.6	1.3
LCL	297	36.640	0.123	0.005	12.3	1.0
MGT	22	2.810	0.128	0.005	12.4	1.0
MSC	12	1.560	0.130	0.004	12.7	1.1
NLP	103	30.790	0.299	0.424	13.6	2.1
PUZ	56	8.150	0.146	0.092	12.8	1.4
Total	792	129.25				

Figure 16. Statistics of `h1` on TPTP Examples, with the Type Proxy Approach

As far as times are concerned, the type proxy approach appears to be roughly 10% faster (30% and 7% respectively in previous experiments). Compensating for startup times, totalling 95 s., the acceleration can be evaluated to 40% (resp. 90% and 30% on previous experiments). Considering that we would like to use the type proxy approach as a first approximation before actual resolution takes place, we would gain some advantage only if total computation time $1/1.4t + 1/\alpha t$ (resp. $1/1.9t + 1/\alpha t$, $1/1.3t + 1/\alpha t$ in previous experiments) were at most t, where t is the time taken by the direct approach: as we have just seen, $1/1.4t$ (resp. $1/1.9t$, $1/1.3t$) is roughly the time taken by the type proxy approach, and we let α be the expected speedup expected from using static soft typing guided by $\mathcal{M}$. It follows that using the type proxy approach for static soft typing can only start to be competitive provided the speedup α is at least 3.5 (resp. 2.1, 4.3).

On the Parrennes examples, figures are much more contrasted, see Figure 17, which should be compared with those of Figure 9. The surprise here is that running the type proxy approach, i.e., deciding S_{proxy},

is sometimes slower than deciding S directly. Here, this is faster for `a1` by 26%, for `a2` by 1%, and slower for the remaining, easier examples. (A previous experiment showed figures of 38%, and 13% respectively. A further previous experiment showed the type proxy approach to be slower in all cases except for `a2`.) Anyway, computation times of the type proxy and the direct approaches are of the same order of magnitude. This is surprising in the `a1` case, since we derived about half as many clauses in the type proxy approach as in the direct approach. Also, we generated only about 6 times less "automaton" clauses, but we still required about the same time (26% less time here; 38% less time and 25% more time our previous experiments) to do this. This can be explained by looking at the number of ne-facts and corresponding frozen clauses. In the case of `a1`, we required to derive 658 ne-facts, and 16 336 clauses were awaken with the direct approach; with the type proxy approach, 1 045 ne-facts were derived, waking up 53 959 clauses.

File	Time (s.)	Memory (Mb)	#clauses generated	subsumed	automaton
`a1`	283.6	160.3	1 443 569	314 771	14 562
`a2`	269.8	150.7	1 346 493	272 201	13 482
`a3`	10.4	32.6	114 711	12 790	3 172
`np`	7.0	22.8	52 289	7 372	2 142
`ns`	9.1	23.2	57 690	7 382	2 321

Figure 17. Running `h1` on the Parrennes Examples, with the Type Proxy Approach

Worse than that: in the examples discussed until now, we have called `h1` in order to find a refutation. In the unsatisfiable examples `a1`, `a2`, and `ns`, finding a model $\mathcal{M}$ of the definite clauses of S_{proxy} requires one to continue to resolve even after some contradiction has been found. (Or to saturate just the subset of definite clauses.) This can be obtained with the `-all` option to `h1`, see Figure 18. While we could discuss the differences with Figure 17, we shall only observe that, even on the large examples `a1` and `a2`, `h1 -all` announces a contradiction, then stops with a saturated set of clauses less than two seconds later. In short, we have observed no significant difference in performance between finding a contradiction and saturating the clause set on the Parrennes examples; and any difference between Figure 17 and Figure 18 only indicates how much variability is to be expected in the conducted experiments. We take this as a vindication that, indeed, the Parrennes examples are

necessarily insensitive to the clause selection strategy, as claimed in Section 4.2, in the sense that randomly changing clause selection does not appear to make `h1` find a contradiction faster, with any amount of luck.

File	Time (s.)	Memory (Mb)	#clauses generated	#clauses subsumed	#clauses automaton
`a1`	358.8	146.6	1 302 746	286 224	14 577
`a2`	389.2	174.6	1 671 334	356 566	15 348
`ns`	9.1	25.4	58 203	7 766	2 266

Figure 18. The Type Proxy Approach, Continuing after the First Contradiction

0ne would still need to evaluate whether the model $\mathcal{M}$ (if any) gotten from the type proxy approach helps in any way guide the search for a refutation or a model in the direct approach. (In case S_{proxy} is unsatisfiable. By the way, on the Parrennes examples, S_{proxy} was found to be satisfiable exactly when S was `a3` or `np`. In other words, the type proxy approximation incurs no loss of precision on these examples.) We have not pursued this. Looking at the figures above, computing $\mathcal{M}$ would have to very significantly reduce the proof search effort through static soft typing against $\mathcal{M}$ before any advantage can be obtained this way. The prospect does not look good.

Out of curiosity, we conducted the following one-shot experiment in 2004. The code of `h1` was hacked up so as to accept an alternating tree automaton $\mathcal{M}$ to guide proof search. We ran this on `a1`. At that time, the optimizations described in Section 5 and later were lacking, and `h1` could not solve `a1`, whether in direct mode or in type proxy mode. So we had another version of `h1` hacked up in such a way that any clause having more than a predefined number of atoms in its body would be reduced by dropping some of its atoms at random. This produced, as above, a model $\mathcal{M}$ for the subset of all definite clauses. By eye inspection, the model looked informative enough, and in any case more informative than the naive pathset approach of Section 4.2. It then turned out that no clause was removed whose body was false in $\mathcal{M}$ that had not already been removed by the naive pathset approach. In other words, static soft typing using the type proxy approach was inefficient, and did not help. Naturally, we can hardly claim any definite conclusion from this experiment.

This is an example where losing precision does not offer any significant performance advantage, and may even be detrimental. We have already seen that approximating a first-order clause set to an ${}_\flat\mathcal{H}_1$ clause

set may result in more difficult problems, and should therefore not be too surprised.

Instead of computing full automata to guide search, one may return to the pathset approach, and improve it only slightly. Instead of computing sets of paths, compute sets of *trees*. This would avoir the loss of precision mentioned above: when we add $P(f(a,b))$ and $P(f(b,a))$ to the current pathset Ψ, we in fact add the four paths $(f,1).(a,0).\Omega$, $(f,2).(b,0).\Omega$, $(f,1).(b,0).\Omega$, and $(f,2).(a,0).\Omega$. In particular, there is no way to tell that we didn't want to add $P(f(a,a))$ or $P(f(b,b))$.

A cure is too replace pathsets by a suitable representation of sets of ground terms, of depth at most k, on an extended signature with a fresh constant Ω denoting any ground term. We let the reader figure out for herself how to adapt the constructions above. The basic construction, replacing pathsets, would be the *treeset*: either Ω, or a map from function symbols f, of arity n, to n-tuples of treesets. We tried this in 2003, and thought at that time that the only way to implement the corresponding fixpoint computation was by using a worklist algorithm. As we saw earlier, this is doomed to failure.

It would therefore, at least in principle, be interesting to implement the naive fixpoint algorithm of Figure 15 on treesets rather than pathsets. Although we didn't conduct any experiment meant to evaluate the treeset approach, we convinced ourselves that this would probably not be worth the trouble. The Parrennes examples are based on the idea [11] that points-to analysis can profitably be described by clauses. In particular, $\texttt{value}(c(s,t))$ means that the value stored at address s is t (possibly). The `csur` analyzer of [11] generates a good deal of addresses, which are denoted by constants (whence the large number of constants in the Parrennes examples), and addresses may point to pointer values containing further addresses. Even if we consider only constants for values of s, t, the treeset approach will necessarily derive a collection of facts $\texttt{value}(c(a_1,a_2))$, where a_1 and a_2 denote addresses such that a_1 points to a_2. In other words, the treeset approach would necessarily build a graph on around 350 nodes (remember there are 349 constants in the examples `a1`, `a2`, `a3`). Worse, some clauses compute operations such as transitive closures, e.g., $\texttt{value}(c(X,Z)) \Leftarrow \texttt{value}(c(X,Y)), \texttt{value}(c(Y,Z))$. (They shouldn't, considering the semantics of pointers. But remember these examples came from an early *buggy* implementation of `csur`.) In effect, the treeset approach would compute the transitive closure of a 350-vertex graph. This is prohibitive. In comparison, the pathset approach just computes over-approximations of the domain and codomain of the $\texttt{value}(c(\_,\_))$ relation. We have therefore also relinquished the treeset approach.

To sum up: naiveté matters.

5. Abbreviating Deep Atoms On-the-Fly

The ϵ-splitting rule is a special case of Riazanov and Voronkov's splitting with naming rule [18]. In turn, this rule can be generalized in the following way. Given any clause

$$H \Leftarrow \mathcal{B}, \mathcal{B}'$$

where we have split the body into two disjoint conjunctions of atoms $\mathcal{B}$ and $\mathcal{B}'$, and where the free variables of $\mathcal{B}$ are $X_1, \ldots, X_m$, create a fresh predicate symbol $T_{\mathcal{B}}$ and a fresh function symbol $\mathtt{abbrv}_{\mathcal{B}}$ of arity m, and replace the above clause by the two clauses

$$\begin{aligned} H &\Leftarrow T_{\mathcal{B}}(\mathtt{abbrv}_{\mathcal{B}}(X_1, \ldots, X_m)), \mathcal{B}' \\ T_{\mathcal{B}}(\mathtt{abbrv}_{\mathcal{B}}(X_1, \ldots, X_m)) &\Leftarrow \mathcal{B} \end{aligned}$$

As for ϵ-splitting, and following the splitting with naming philosophy, we reuse the same predicate $T_{\mathcal{B}}$ and the same function symbol $\mathtt{abbrv}_{\mathcal{B}}$ when we encounter the same conjunction $\mathcal{B}$. It is even profitable to rename variables in $\mathcal{B}$ so as to maximize the chances to reuse the same symbols.

We call this rule the *abbreviation* rule.

This rule is sound. Using Bachmair and Ganzinger's general redundancy elimination criterion, this rule is complete provided the two generated clauses are smaller than the premise $H \Leftarrow \mathcal{B}, \mathcal{B}'$, in a so-called *admissible* clause ordering [2]. Given our ordering $\succ$ on atoms, an admissible clause ordering is given by, first, comparing literals by $\pm A \succ \pm' B$ if and only if $A \succ B$, or $\pm$ is $-$, $\pm'$ is $+$, and $A \succeq B$; second, by considering clauses as multisets of literals, compared by the multiset extension $\succ^{mul}$ of $\succ$.

One way to achieve completeness is to extend the $\succ$ ordering on atoms so that $P(t) \succ T_{\mathcal{B}}(t') \succ q_B$ for each predicate symbol already present in the original clause set, for each ne-fact q_B, and for every symbol $T_{\mathcal{B}}$. Then completeness obtains if $\mathcal{B}$ contains at least one atom not headed by a T or q_B symbol, and $H, \mathcal{B}'$ contains at least an atom not headed by a T or q_B symbol.

The most classical use of such rules is when $\mathcal{B}$ is a non-empty conjunction of atoms that has no free variable in common with the rest $H \Leftarrow \mathcal{B}'$ of the clause $H \Leftarrow \mathcal{B}, \mathcal{B}'$. This allows one to simulate splitting in the general case [18]. This was known in the field of logic program transformation as the rule of *definition introduction* [17]. The latter was notably used to obtain decidability results for inductive definitions over language expressions, and quasi-cs logic programs [14].

The particular case of the abbreviation rule that we will be interested in is the case where $\mathcal{B}$ is reduced to a single deep atom $P(t)$. We say

that $P(t)$ is *deep* if and only if it is not shallow; $P(t)$ is *shallow* if and only if t is a variable of the application $f(Y_1, \ldots, Y_n)$ of a function symbol f to variables Y_1, ..., Y_n, not necessarily distinct.

It is probably not clear what can be gained from such a rule. We address this in a jiffy. Meanwhile, let us formalize it.

First, we need to find a canonical term $\widehat{t}$ for each term t, so that if t and t' are renamed versions of each other, then $\widehat{t} = \widehat{t'}$. This can be obtained as follows. Fix a countable enumeration of variables X_1^0, ..., X_m^0, ..., once and for all. Standardize the set of free variables of each term t by enumerating the free variables of t from left to right without duplication, say X_1, ..., X_m; e.g., enumerate the free variables of $f(X, g(X, f(Y, X), Y, Z, X))$ as X, Y, Z. Then replace X_1, ..., X_m by X_1^0, ..., X_m^0, that is, define $\widehat{t}$ as $t\{X_1 := X_1^0, \ldots, X_m := X_m^0\}$.

Let $\mathcal{P}$ be the set of predicate symbols and $\mathcal{F}$ the set of function symbols of the original clause set. Let $\mathcal{Q}$ be the set of splitting symbols q_B. For any atom $P(t)$ where $P \in \mathcal{P}$ and t is built from function symbols taken from $\mathcal{F}$, create a predicate symbol $T_{P(\widehat{t})}$ outside $\mathcal{P} \cup \mathcal{Q}$, and a function symbol $\texttt{abbrv}_{P(\widehat{t})}$ outside $\mathcal{F}$. We assume that the functions $P(\widehat{t}) \mapsto T_{P(\widehat{t})}$ and $P(\widehat{t}) \mapsto \texttt{abbrv}_{P(\widehat{t})}$ are one-to-one.

DEFINITION 2. The *abbreviation of deep atoms* rule is the abbreviation rule restricted to the case where $\mathcal{B}$ is a single deep atom $P(t)$. Concretely, this is the rule that replaces any clause

$$H \Leftarrow P(t), \mathcal{B}$$

where $P(t)$ is deep, $P \in \mathcal{P}$, all function symbols of t are in $\mathcal{F}$, and there is at least one atom of the form $Q(u)$ with $Q \in \mathcal{P}$ in $H \Leftarrow \mathcal{B}$, by the two clauses

$$H \Leftarrow T_{P(\widehat{t})}(\texttt{abbrv}_{P(\widehat{t})}(X_1, \ldots, X_m)), \mathcal{B}$$
$$T_{P(\widehat{t})}(\texttt{abbrv}_{P(\widehat{t})}(X_1, \ldots, X_m)) \Leftarrow P(t)$$

where X_1, ..., X_m are the free variables of t from left to right, enumerated without duplication.

This is sound, and complete by Bachmair and Ganzinger's general redundancy elimination criterion, provided we take an ordering $\succ$ such that $P(t) \succ T_{Q(\widehat{u})}(t') \succ q_B$ for every $P \in \mathcal{P}$, as discussed above.

To illustrate the speedups that can be achieved with this rule, run `h1` with the `-no-deep-abbrv` option, which disables it, on the Parrennes examples.

This is summed up in Figure 19. The $> x$ entries mean that the `h1` process had to be killed because it exceeded the allotted space limit of

File	Time (s.)	Memory (Mb)	#clauses generated	subsumed	automaton
a1	> 137.6	> 391	> 1 354 191	> 165 245	> 1 531
a2	> 175.9	> 391	> 1 481 786	> 205 391	> 2 661
a3	> 87.9	> 391	> 1 054 217	> 37 254	1 321
np	355.5	298.5	1 249 194	357 266	2 052
ns	360.2	241.2	1 262 224	360 445	2 112

Figure 19. No Deep Abbreviations on the Parrennes Examples

400 000Kb= 391Mb. We still report how much was derived when the process was killed. E.g., on a1, when the h1 process was killed, it had produced 1 531 "automaton" clauses.

None of the problems a1, a2, a3 could be solved within the space limit we required, without deep abbreviation. In fact, all three required more than 400 000Kb, and were far from solved. On a3, the h1 process without deep abbreviation was killed when only one fourth of the "automaton" clauses were produced. On a1 and a2 this happened while only one over 55, resp. 32, was produced.

The other two problems, np and ns, were solved without deep abbreviation, however in roughly 120-140 times more time and roughly 11-15 times more memory. Many more clauses were generated: roughly 45-50 times more on np and ns. On the other hand, the number of "automaton" clauses was *reduced* by about 45-50%. The latter may seem paradoxical, however note that deep abbreviation not only generates new function symbols, but also, necessarily, new alternating automaton clauses. Not using the deep abbreviation rule necessarily produces smaller sets of "automaton" clauses.

These drastic improvements in time and space are obtained by applying the deep abbreviation rule in surprisingly very few places. Indeed, the results of Figure 9, with deep abbreviation enabled, were obtained while generating only 178 $T_{P(\hat{t})}$ symbols in the a1 case, 178 also for a2, 177 for a3, 73 for np, and 71 for ns.

Let us explain these figures, which are admittedly rather surprising. Essentially, they tend to support the view that deep abbreviation is a miracle: it costs nothing, and gives you speedups of several orders of magnitude.

We obtained the explanation by looking at a trace of the proof obtained by h1 on the a1 example, and comparing with a trace of the (partial) proof obtained by h1 without the deep abbreviation rule. (Out

of curiosity, the complete proof obtained by `h1` with deep abbreviation on `a1` contains 9 460 000 steps. The trace occupies 119Mb in a `gzip`-compressed file, which takes 5min. 40 just to decompress, to a 3 Gb file. The `h1logstrip` utility can be used to extract the subset of clauses actually used in deriving the empty clause, and yields a 2 205 step proof, which can be translated using the `h1trace` utility into a 828 step unit resolution proof.)

Look at what happens without the deep abbreviation rule. At some point during proof search, some deep atom $P(t)$ may occur in the bodies of several (non-frozen) clauses, say $H_1 \Leftarrow P(t), \mathcal{B}_1, \ldots, H_\ell \Leftarrow P(t), \mathcal{B}_\ell$. In the case of `a1`, we concentrated on a given deep atom $P(t)$ taken arbitrarily, and realized that, at the point where `h1` exceeded the memory limit, there were roughly 10 000 clauses containing $P(t)$; i.e., $\ell \approx 10\,000$.

Since the clauses $H_1 \Leftarrow P(t), \mathcal{B}_1, \ldots, H_\ell \Leftarrow P(t), \mathcal{B}_\ell$ are not frozen, i.e., do not contain any splitting symbol q_B, our selection function will necessarily select $P(t)$, possibly among others. This is because $P(t)$, being deep, is not simple. These clauses can therefore only be used as main premises. Write t as $f(t_1, \ldots, t_k)$. The corresponding side premises are either the universal clause $P(X)$, which is easily dealt with, or a collection of alternating automaton clauses $P(f(X_1, \ldots, X_k)) \Leftarrow B_{11}(X_1), \ldots, B_{1k}(X_k), \ldots, P(f(X_1, \ldots, X_k)) \Leftarrow B_{n1}(X_1), \ldots, B_{nk}(X_k)$. Assuming, for the sake of simplicity, that we resolve only on $P(t)$ (only $P(t)$ is selected). The resolvents are $H_i \Leftarrow B_{j1}(t_1), \ldots, B_{jk}(t_k), \mathcal{B}_i$. There may be up to ℓn of them. More importantly, $t_1, \ldots, t_k$ may themselves be deep, so that more resolvents can be generated from each of the resolvents we have computed.

In the presence of deep abbreviation, the ℓ clauses $H_1 \Leftarrow P(t), \mathcal{B}_1, \ldots, H_\ell \Leftarrow P(t), \mathcal{B}_\ell$ will generate the m abbreviated clauses

$$\begin{aligned} H_1 &\Leftarrow T_{P(\widehat{t})}(\mathtt{abbrv}_{P(\widehat{t})}(X_1, \ldots, X_m)), \mathcal{B}_1 \\ &\ldots \\ H_\ell &\Leftarrow T_{P(\widehat{t})}(\mathtt{abbrv}_{P(\widehat{t})}(X_1, \ldots, X_m)), \mathcal{B}_\ell \end{aligned} \tag{11}$$

plus just one clause defining $T_{P(\widehat{t})}$ and $\mathtt{abbrv}_{P(\widehat{t})}$:

$$T_{P(\widehat{t})}(\mathtt{abbrv}_{P(\widehat{t})}(X_1, \ldots, X_m)) \Leftarrow P(t) \tag{12}$$

The n alternating automaton clauses $P(f(X_1, \ldots, X_k)) \Leftarrow B_{11}(X_1), \ldots, B_{1k}(X_k), \ldots, P(f(X_1, \ldots, X_k)) \Leftarrow B_{n1}(X_1), \ldots, B_{nk}(X_k)$ can now only be resolved with (12). Note that since t is deep, $P(t)$ is selected. This implies, first, that we can indeed produce the n resolvents with the n alternating automaton clauses above. More importantly, this disallows resolving (12) against any of the ℓ clauses (11).

The ℓ clauses (11) will only be used as main premises when enough alternating automaton clauses have been resolved against (12). E.g., let t be $f(X, g(X, f(Y, X), Y, Z, X))$, and consider the alternating automaton clauses

$$P(f(X_1, X_2)) \Leftarrow P_1(X_1), P_2(X_2) \quad (13)$$
$$P_2(g(Y_1, Y_2, Y_3, Y_4, Y_5)) \Leftarrow P_3(Y_1), P_4(Y_2), P_5(Y_3), P_6(Y_4), P_7(Y_5) \quad (14)$$
$$P_4(f(Z_1, Z_2)) \Leftarrow P_8(Z_1), P_9(Z_2) \quad (15)$$

Then the only possible resolution steps between the latter and the clauses (11) and (12) are

$$\dfrac{\dfrac{\dfrac{(12)}{T_{P(\hat{t})}(\mathtt{abbrv}_{P(\hat{t})}(X_1, \ldots, X_m)) \Leftarrow P_1(X), P_2(g(X, f(Y, X), Y, Z, X))}\,(13)}{T_{P(\hat{t})}(\mathtt{abbrv}_{P(\hat{t})}(X_1, \ldots, X_m)) \Leftarrow P_1(X), P_3(X), P_4(f(Y, X)), P_5(Y), P_6(Z), P_7(X)}\,(14)}{T_{P(\hat{t})}(\mathtt{abbrv}_{P(\hat{t})}(X_1, \ldots, X_m)) \Leftarrow P_1(X), P_3(X), P_8(Y), P_9(X), P_5(Y), P_6(Z), P_7(X)}\,(15)$$

and only then we can resolve the latter against each of the ℓ clauses (11).

Using the deep abbreviation rule therefore implements some *sharing* of computations: all ℓn resolution steps that were needed without deep abbreviation are summed up by just n resolution steps with a single clause (11). Remember that ℓ was of the order of 10 000, at least, on the `a1` example. On this particular example, we conclude that the speedup gained by deep abbreviation is at least 10 000.

This is a general fact for ${}_\flat\mathcal{H}_1$ clause sets. Observe that ordered resolution with selection may produce a number of clauses that is exponential in the size of the initial ${}_\flat\mathcal{H}_1$ clause set S_0. More precisely, this may produce at most

$$(1 + 2^{n_p} + n_p + n_p n_f) 2^{n_p N} e 2^{n_f a}$$

clauses that are not alternating automaton clauses, and at most $n_p n_f 2^{n_p a}$ alternating automaton clauses, where n_p is the number of predicate symbols in S_0, n_f is the number of function symbols, a is the maximal arity of function symbols, and N is the number of distinct subterms of terms in S_0 [9, proof of Theorem 6]. On the other hand, it is shown in op.cit. that all terms that occur in non-alternating automaton clauses must be subterms of terms in S_0. In particular, there are at most N of them. One easy consequence is that we can expect the number of non-alternating automaton clauses that contain the same given atom $P(t)$ to grow to about

$$\frac{(1 + 2^{n_p} + n_p + n_p n_f) 2^{n_p N} e 2^{n_f a}}{n_p N}$$

which is clearly an exponential in the size of S_0.This is all the more critical as $n_p N$ grows, in particular as there are deep atoms in S_0. For large clause sets containing deep atoms, deep abbreviation is bound to yield dramatic improvements in performance.

Not every ${}_\flat\mathcal{H}_1$ problem reaches the exponential upper bound in the numerator above. But, even on problems in the smaller class $\mathcal{H}_3$ (we shall argue later that the Parrennes examples, despite their apparent difficulty, are in fact close to an $\mathcal{H}_3$ clause set), the numerator would be cubic in the size of the input clauses, and we can still expect the average number of non-alternating automaton clauses that contain the same given atom $P(t)$ to be quadratic in N. Since `h1` produced about 180 abbreviation symbols on `a1`, there were at least 180 subterms in the input clause set `a1`, i.e., $N \geq 180$. The square of this is 32 400, which confirms our early speedup estimate on `a1` of at least 10 000.

This much explains why deep abbreviation may result in some speed-up, but remember that `a1`, `a2`, `a3` failed by exceeding our fixed *space* limit, not the time limit. One paradox is that running `h1` without deep abbreviation fails by consuming more that 391Mb while generating only 1.35 million clauses, with about 1.2 million kept, whereas `h1` with deep abbreviation succeeds, consumes only 262.1Mb while generating 2.9 million clauses, with about 2.2 million kept. This is more mysterious. Our best explanation is that all resolvents of the defining clause (12) with automata clauses share the same head $T_{P(\widehat{t})}(\mathtt{abbrv}_{P(\widehat{t})}(X_1, \ldots, X_m))$, and have no additional literals in the body. Remember that `h1` is coded in HimML, a language where all equality-admitting data is hash-consed. So `h1` automatically shares identical parts of clauses. Any resolvent obtained this way, say $T_{P(\widehat{t})}(\mathtt{abbrv}_{P(\widehat{t})}(X_1, \ldots, X_m)) \Leftarrow P_1(t_1), \ldots, P_n(t_n)$, would give rise, without deep abbreviation, to resolvents of the form $H_i \Leftarrow P_1(t_1), \ldots, P_n(t_n), \mathcal{B}_i$, $1 \leq i \leq \ell$, with different heads H_i and different additional literals $\mathcal{B}_i$, therefore reducing the impact of hash-consing.

So the combination of hash-consing and deep abbreviation is likely to be responsible for the dramatic space advantage deep abbreviation offers us in `h1`. Quantifying this advantage, i.e., predicting memory usage is rather arduous, much as in the case of BDDs [3]—where space usage is essentially unpredictable, and is in any case completely uncorrelated to the size of the input problem [5].

This unpredictability can be illustrated in our setting. One should observe that in other situations, no or little sharing actually happens: e.g., `h1` run on the `PUZ052-1` example from the TPTP library runs out of memory (even in the presence of the deep abbreviation rule, which it uses only once), although only 186 162 non-"automaton" clauses and

57 994 "automaton" clauses were generated, and 12 171 erased, for a total of only about 232 thousand kept clauses.

Evaluating the effect of deep abbreviation on the TPTP examples was again done by running `h1` on these examples, disabling deep abbreviation again. See Figure 20.

Cat.	#problems solved	Time (s.)			Memory (Mb)	
		Total	Avg.	St.Dev.	Avg.	St.Dev.
COM	6	0.810	0.135	0.015	11.8	0.3
FLD	279	46.250	0.166	0.015	13.6	1.6
KRS	17	2.200	0.129	0.011	12.2	1.0
LCL	297	36.610	0.123	0.005	12.3	1.0
MGT	22	2.810	0.128	0.004	12.4	1.0
MSC	12	1.490	0.124	0.008	12.2	0.8
NLP	103	69.250	0.672	2.559	13.6	2.1
PUZ	57	13.130	0.230	0.665	12.8	1.9
Total	793	172.55				

Figure 20. Statistics of `h1` on TPTP Examples, without Deep Abbreviation

Judging from these results, it would seem that deep abbreviation is in fact counter-productive on average. One *more* example is dealt with by disabling deep abbreviation. (Again, this is not significant: if one problem takes 5 minutes, up to a typical variation of 10 seconds around this mean value, it will be killed or not essentially at random.) But using deep abbreviation results in an overall 21% speedup (62% if we take startup times into account). This is not really significant: as we have already noticed, times should be taken with a rough $\pm 15\%$ error margin.

However, look at Figure 21, where we have collected the number of TPTP examples (on the y-axis) requiring n instances of the deep abbreviation rule, for $n = 0, 1, 2, \ldots$ (on the x-axis). (The x values 0.25 and 0.5 are meaningless, but are produced automatically by `gnuplot`: the leftmost column corresponds to 0 abbreviation [59 problems], the next one corresponds to 1 abbreviation [221 problems], etc. Note that the x-axis is again plotted with a logarithmic scale. The next peaks are $n = 3$, with 284 problems, and $n = 4$ with 114 problems.) The conclusion is clear: almost no abbreviation of deep atoms ever occurs in the TPTP examples. In other words, almost all terms in bodies of

clauses in these examples are shallow. It is therefore unsurprising that deep abbreviation does not yield much improvement on these examples.

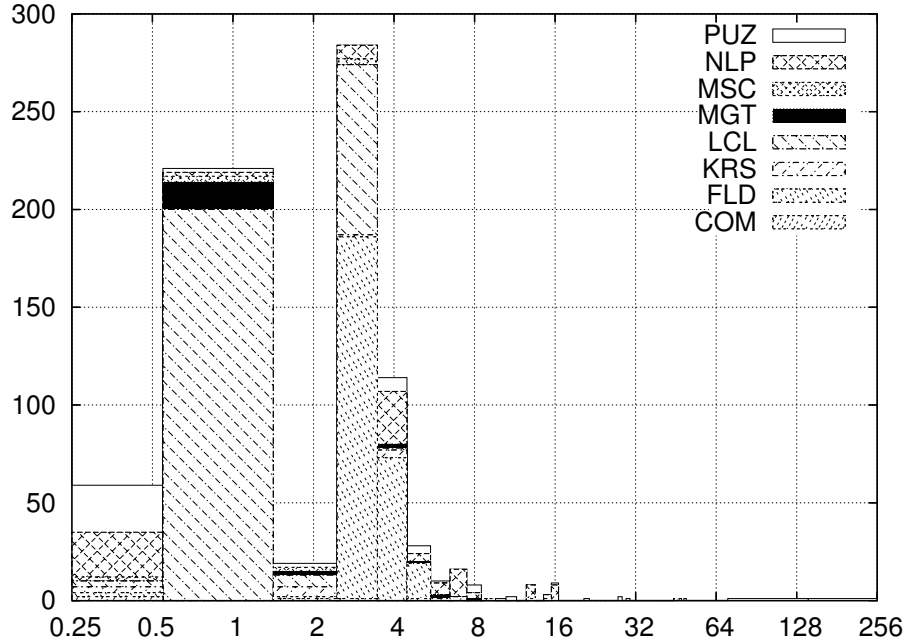

Figure 21. Breakdown of TPTP Examples According to Number of Abbreviated Deep Atoms

We end this section with two remarks. First, introducing abbreviation symbols requires one to recompute Ψ_∞. As for splitting symbols, this is easy: for each defining clause (12) $T_{P(\widehat{t})}(\mathtt{abbrv}_{P(\widehat{t})}(X_1,\ldots,X_m)) \Leftarrow P(t)$, let ϱ be such that $[] \vdash t : \Psi_\infty(P) \Rightarrow \varrho$, then add $\{T_{P(\widehat{t})} \mapsto \mathtt{abbrv}_{P(\widehat{t})}(X_1,\ldots,X_m)\varrho\}$ to Ψ_∞. If ϱ does not exist, then naive static soft typing will have erased the parent clause $H \Leftarrow P(t), \mathcal{B}$.

Second, one might wonder whether other variants of the abbreviation rule could offer similar speedups. It is probably futile to abbreviate atoms that are not deep, because this would amount to abbreviate shallow atoms $P(t)$ by atoms $T_{P(\widehat{t})}(\mathtt{abbrv}_{P(\widehat{t})}(X_1,\ldots,X_m))$ that are no deeper. But we might design rules that abbreviate groups of more than one atom. However, there is some reason to believe that this won't bring as many benefits. Recall that the efficiency of abbreviation is proportional to the expected number of clauses containing any given atom $P(t)$. If we are to abbreviate, say, unordered pairs $P_1(t_1), P_2(t_2)$ of atoms, this expected number will be an exponential in N divided by about N^2 in the worst case—about N^3 over N^2 in the $\mathcal{H}_3$ case. There

may still be some opportunity to gain some speedup here, but they will occur N times more rarely. When $N \geq 180$ as in the `a1` example, it is probably not worth the trouble; and when N is small, as we have seen on the TPTP examples, abbreviation is useless anyway.

5.1. Removing Alternation

We still conducted an experiment which may give an idea of the efficiency of another variant of the abbreviation rule. Our purpose here was not in evaluating variants of the abbreviation rule per se. Rather, we originally wished to implement a rule that would convert alternating automata to non-deterministic automata—which we take here as meaning clauses (1) where each block $B_i(X_i)$ contains at most one predicate symbol. This is implemented as follows.

DEFINITION 3. The *abbreviation of non-trival blocks* rule is the abbreviation rule restricted to the case where $\mathcal{B}$ is an non-trivial ϵ-block, i.e., an ϵ-block $B(X)$ containing at least two predicate symbols. Concretely, this is the rule that replaces any clause

$$H \Leftarrow P_1(X), \ldots, P_n(X), \mathcal{B}$$

where $n \geq 2$, $P_1, \ldots, P_n \in \mathcal{P}$, X is not free in $H \Leftarrow \mathcal{B}$, and there is at least one atom of the form $Q(u)$ with $Q \in \mathcal{P}$ in $H \Leftarrow \mathcal{B}$, by the two clauses

$$H \Leftarrow T_{P_1(X_1^0),\ldots,P_n(X_n^0)}(X), \mathcal{B} \tag{16}$$

$$T_{P_1(X_1^0),\ldots,P_n(X_n^0)}(X) \Leftarrow P_1(X), \ldots, P_n(X) \tag{17}$$

Call $T_{P_1(X_1^0),\ldots,P_n(X_n^0)}$ an *intersection predicate*, and (17) its *defining clauses*.

Recall that X_1^0 is the first in the series of variables that we used to define the canonical form $\widehat{t}$ of terms t. Note that we do not introduce any $\texttt{abbrv}_{P_1(X_1^0),\ldots,P_n(X_n^0)}$ function symbol, which would have only one argument anyway. (So, formally, this is not a special case of the abbreviation rule. It differs rather deeply from the abbreviation rule in that $T_{P_1(X_1^0),\ldots,P_n(X_n^0)}(X)$ will in general not be selected in clause (16).) We have implemented the abbreviation of non-trivial blocks rule restricted to the case where the parent clause $H \Leftarrow P_1(X), \ldots, P_n(X), \mathcal{B}$ is an alternating automaton clause. This is enough to convert alternating tree automata to non-deterministic tree automata.

The results on the TPTP examples are shown in Figure 22. Less problems were solved (22 remained unsolved, compared to 16), in more time: 3.6 more time, 8.8 if we take startup times into account.

Cat.	#problems solved	Time (s.) Total	Avg.	St.Dev.	Memory (Mb) Avg.	St.Dev.
COM	7	1.790	0.256	0.262	15.2	2.2
FLD	279	46.140	0.165	0.015	13.8	1.5
KRS	17	2.180	0.128	0.012	13.3	1.3
LCL	297	36.890	0.124	0.005	12.4	1.1
MGT	22	2.780	0.126	0.006	13.4	1.3
MSC	12	1.500	0.125	0.006	13.2	1.3
NLP	96	414.460	4.317	26.154	20.3	38.5
PUZ	56	8.180	0.146	0.094	13.3	1.7
Total	786	513.92				

Figure 22. Statistics of `h1` on TPTP Examples, with Abbreviation of Non-Trivial Blocks

This should, in fact, not be too surprising. Remember that conversion from alternating tree automata to non-deterministic tree automata takes exponential time in general. In fact, while testing the emptiness of alternating tree automata is DEXPTIME-complete, testing the emptiness of non-deterministic tree automata is doable in polynomial-time; and P≠DEXPTIME: this conversion is provably not polynomial-time.

We counted the number of intersection predicates $T_{P_1(X_1^0),\ldots,P_n(X_n^0)}$ generated on each TPTP problem, and plotted the number of problems that required us to generate $n = 0, 1, 2, \ldots$ intersection predicates, see Figure 23. To this end, we used the `-no-alternation` option to `h1`, which enables the use of the abbreviation of non-trivial blocks rule on generated alternating automaton clauses. Note that only problems on which `h1` with the `-no-alternation` option succeeded are reported.

The figure should make it clear that, among the problems that `h1` still succeeded in solving with the abbreviation of non-trivial blocks rule, a huge majority (700 among 786) never used the rule. It is a typical property of the cubic-time subclass $\mathcal{H}_3$ of $\mathcal{H}_1$ that the abbreviation of non-trivial blocks rule never applies to $\mathcal{H}_3$ clauses, because no variable can occur more than once in the body of $\mathcal{H}_3$ clauses. This suggests that the great majority of the TPTP problems is close to $\mathcal{H}_3$ in this sense. Together with our earlier remark that almost no deep atom occurs in the TPTP examples, this indicates that most of the TPTP examples are, in fact, easy.

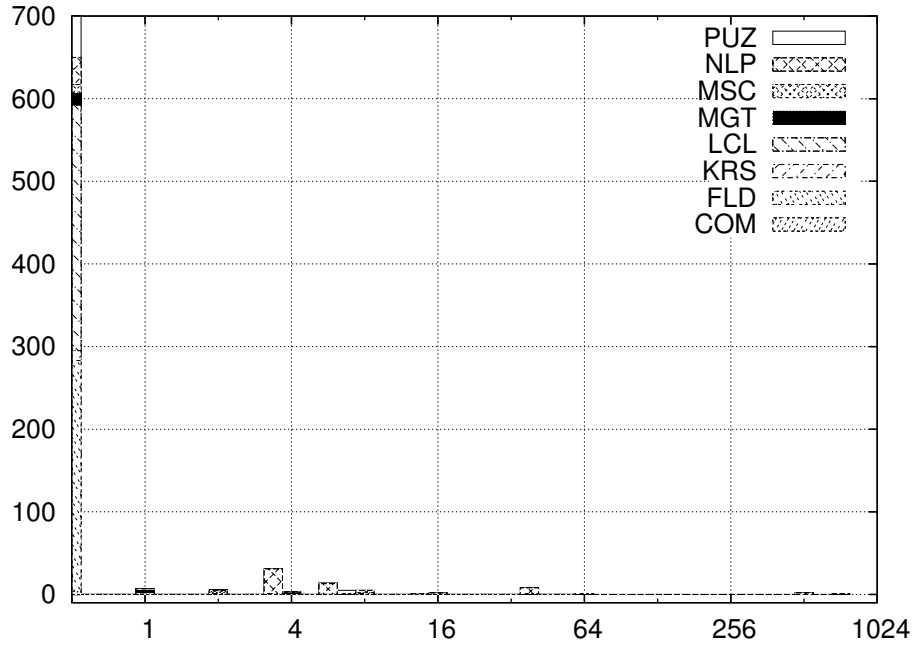

Figure 23. Breakdown of TPTP Examples According to Number of Intersection Predicates

The remaining TPTP problems, which were solved by `h1` but failed in the presence of the abbreviation of non-trivial blocks rule, had generated 600 intersection predicates on average, with a standard deviation of 250, before failing.

Curiously, two problems were much easier to solve with the abbreviation of non-trivial blocks rule, `PUZ038-1` and `COM006-1`, which terminated in 0.8 s. and 0.9 s. respectively, generated 4 325, resp. 22 924 clauses, erasing 1 179, resp. 10 904 clauses, and producing 197, resp. 110 "automaton" clauses. The first applied the abbreviation of non-trivial blocks rule to generate 5 intersection predicates, the second generated 62 intersection predicates. The `PUZ038-1` case seems to be pure luck, and is hard to reproduce. The `COM006-1` case is stranger: while a contradiction was found, the 462 step refutation does not use any of the 5 defining clauses produced. The reason is that none of the defining clauses resolves again any group of side premises that are ever produced; apparently this behavior is not simulated by `h1` without the abbreviation of non-trivial blocks rule.

The results on the Parrennes examples are shown in Figure 24. Up to inevitable variations, times and memory consumption are about the same as in Figure 1. The number of clauses generated is the same, up

to a small 2% difference for `a1`, `a2`, `a3`. More clauses are generated in the case of `np` (18%) and `ns` (11%). But the most notable difference is in the number of "automaton" clauses, which grew by 17% for `a1`, 18% for `a2`, 8% for `a3`, 17% for `np`, and 16% for `ns`. Indeed, necessarily new alternating automaton clauses must be generated whose head starts with intersection predicate symbols.

File	Time (s.)	Memory (Mb)	#clauses generated	#clauses subsumed	#clauses automaton	#intersection predicates
`a1`	324.2	322.8	2 982 708	687 122	98 977	93
`a2`	310.6	251.5	3 155 531	721 101	100 856	93
`a3`	7.5	30.7	86 255	18 466	6 063	93
`np`	4.4	20.8	29 644	5 308	3 473	51
`ns`	3.8	21.1	31 103	5 417	3 645	49

Figure 24. Abbreviation of Non-Trivial Blocks on the Parrennes Examples

We conclude this section by observing that deep abbreviation is remarkably successful on ${}_\flat\mathcal{H}_1$ clause sets. We have demonstrated that this was due to a cardinality argument: it implements sharing of a polynomial number of atoms among possibly exponentially many clauses. We have also observed that apparently similar rules such as abbreviation of non-trivial blocks could not be as efficient, or could even be detrimental to proof search. Now, it is an interesting question whether deep abbreviation could fare as well on general clause sets, where our cardinality argument above cannot apply.

6. Resolving on Fully-Defined Predicates

The `h1` tool is a blend of many algorithms, data structures, tricks and optimizations. It is futile to describe them all; at this level of detail, the source code would probably be the best documentation. We would like, nonetheless, to address an optimization that appears to be new, whose purpose is not to accelerate proof-search but to save space, full definition subsumption. To address this, we have to describe the implementation of `h1` in slightly more detail.

The `h1` tool rests on a classic loop. It operates on two variables containing clause sets: Wo contains the set of *worked off* clauses, and is initially empty; Us is a priority queue of *usable* clauses, and is initialized to the initial set of clauses, minus all tautologies. Priorities, a.k.a.,

weights, are assigned to each clause in Us so that clauses with smallest weights are selected first.

The basic loop—not quite the `h1` loop, but close—works as the following pseudo-code (we use "resolvent" to mean the conclusion of the ordered resolution with selection rule):

```
 1.while Us ≠ ∅ {
 2.      pick some clause C with smallest weight from Us,
 3.           and remove it from Us;
 4.      if C is the empty clause ⊥ then return "Unsatisfiable";
 5.      if C is subsumed by some clause from Wo then continue;
 6.      remove all clauses subsumed by C from Wo;
 7.      for each resolvent C' of C with some clauses in Wo,
 8.         for each clause H ⇐ B obtained from C' by ε-splitting
                                                 and deep abbreviation,
 9.            if H ⇐ B is not a tautology
10.              and Ψ∞ ⊩ B ⇒ ϱ for some ϱ
11.              then add H ⇐ B to Us;
12.      add C to Wo;
13.}
14return "Satisfiable";
```

This loop is relatively naive by today's standards. E.g., no clause in Us is ever removed before it is picked at line 2, or is ever used to subsume the clause C gotten at line 2.

The weight assigned to C when it is inserted into Wo at line 12 is simple, and the result of trial and error: alternating automaton clauses have weight 0, goal clauses $\bot \Leftarrow \mathcal{B}$ are assigned weight 1, clauses of the form $q_B \Leftarrow \mathcal{B}$, where q_B is a splitting symbol, and clauses whose body contain some splitting symbol, get weight 2; clauses whose head starts with an abbreviation symbol $T_{P(\widehat{t})}$ have weight 3; clauses of the form $P(X) \Leftarrow \mathcal{B}$ have weight 4. The weight of all other clauses $H \Leftarrow \mathcal{B}$ is 4 plus the size of the body $\mathcal{B}$. This reflects the order in which we prefer to consider each kind of clause. However, this is again rather naive by today's standards. In particular, we only select clauses by weight, not depth. This is also far from the sophistication displayed in recent studies [19]. However, remember our goal here is not so much in competitive testing (see Section 1.1).

One improvement to the above algorithm, implemented in `h1`, replaces line 12 by: add C to Wo if C is not strictly subsumed by any of the clauses $H \Leftarrow \mathcal{B}$ added to Us at line 11. This implements a form of resolution subsumption, see [2].

At any given stage of this algorithm, say that a predicate symbol P is *fully defined* if and only if all "automaton" clauses with head starting

with P that will ever be produced are already in Wo. In other words, if and only if $P(X)$ will not be added later to Wo, nor any alterating automaton clause of the form $P(f(X_1, \ldots, X_k)) \Leftarrow B_1(X_1), \ldots, B_k(X_k)$. This can be estimated as follows. Maintain an additional table Δ, mapping each predicate symbol P to the number of clauses of the form $P(t) \Leftarrow \mathcal{B}$ for some t and $\mathcal{B}$, that lie in Us, or that are in Wo but are neither alternating automaton nor universal clauses. It is easy in principle (although it is hard to implement without bugs, in our experience) to maintain these counts, by a reference counting style algorithm. If $\Delta(P)$ drops to 0, then P is fully defined.

The point in fully defined symbols is that they arise naturally from the approximation procedure of Section 3.2. The predicate symbol Q in rule (7), notably, will be classified as fully defined as soon as the defining clause $Q(t) \Leftarrow \mathcal{B}$ is picked at line 2. This is natural: $Q(t)$ is indeed completely defined by the latter clause, as an abbreviation for $\mathcal{B}$. The deep abbreviation symbols $T_{P(\widehat{t})}$ will also profit from this rule, at some later time.

Assume that P is fully defined. If the clause C picked at line 2 is of the form $H_0 \Leftarrow P(t_0), \mathcal{B}_0$ where $P(t_0)$ is selected, then (assuming, for the sake of simplicity, that we resolve on $P(t)$ only, amongst all possible atoms in the body $P(t_0), \mathcal{B}_0$) all the resolvents between C as main premise and side premises $P(t_i) \Leftarrow \mathcal{B}_i$ (which must be universal or alternating automaton clauses) that will ever be generated must be generated at line 7, and not at any later turn of the loop. We can therefore remove C after lines 7-11; in other words, in this case, we do not add C to Wo at line 12. We call this removal rule *straight full definition subsumption*.

There is another, symmetric case where knowing that P is fully defined helps, namely when C at line 2 is the last "automaton" clause to consider before P is fully defined. Formally, when $\Delta(P) = 1$ just before line 2, where C is an "automaton" clause with a head of the form $P(s)$ for some s. Then all the clauses $H_0 \Leftarrow P(t_0), \mathcal{B}_0$ against which C will be resolved at line 7 can be removed from Wo. Call this removal rule *backward full definition subsumption*.

We go a bit further, and choose to resolve on atoms starting with a fully defined symbol prioritarily. In other words, we exploit the freedom in the definition of the selection function that we had observed in Section 3.1. Here is the real definition of our selection function, in the case 2. where the body $\mathcal{B}$ of the given clause contains at least some non-simple atom: if some non-simple atom $P(t)$ in $\mathcal{B}$ is such that P is fully defined, then select all non-simple atoms $P(t)$ with P fully defined from $\mathcal{B}$; otherwise, select all non-simple atoms from $\mathcal{B}$. This is implemented in `h1`.

Contrarily to other rules of `h1`, we have not made a detailed comparison between using the above selection function, with straight and backward full definition subsumption, and not using it. This indeed requires quite some recoding effort. However, we can get some intuition as to how much straight and backward full definition subsumption helps, by looking at `h1` statistics. On the Parrennes examples, figures are shown in Figure 25.

File	#clauses generated	#clauses removed			
		forward subsumption	backward subsumption	full def. subsumption straight	backward
`a1`	2 912 977	593 550	12 709	101 557	0
`a2`	3 063 812	619 287	12 554	106 671	0
`a3`	83 159	13 574	510	5 139	0
`np`	25 062	2 944	20	1 639	0
`ns`	28 110	3 019	16	1 630	0

Figure 25. Full Definition Subsumtion on the Parrennes Examples

The effect of full definition subsumption is not completely negligible: we remove about 3% of all subsumed clauses this way in the hard examples `a1` and `a2`, about 30% in `a3`, about half in `np`, and about 20% in `ns`. However, backward full definition subsumption is totally useless.

Note finally that full definition subsumption is useful to save space, but does not help as far as speed is concerned. With ordinary subsumption, removing subsumed clauses also decreases the number of (useless) subsequent resolution steps that can be done. No such thing happens, by definition, with full definition subsumption.

7. Conclusion

We have described three optimizations that contribute to the good general performance of `h1`. While full definition subsumption is likely to save some space, and naive static soft typing contributes to speedups of the order of 30% (naiveté matters!), the single real rule that makes `h1` efficient is deep abbreviation. We have shown that this rule implemented, at negligible cost, a form of sharing both of clauses and of computations. Speedups of at least 4 orders of magnitude could then be obtained this way on clause sets with deep atoms. By investigating how this could be at all possible, we realized that, in ${}_\flat\mathcal{H}_1$ clause sets,

only polynomially many atoms could ever be produced, but up to an exponential number of clauses could be generated. Consequently, some atoms must be shared in the proportion of an exponential divided by a polynomial. While this is true for ${}_\flat\mathcal{H}_1$, it is an interesting question whether a similar phenomenon is bound to happen on general first-order clause sets, and whether deep abbreviation would profit to general-purpose theorem provers.

Acknowledgements.

We wish to thank Helmut Seidl for several interesting discussions, Julien Olivain for raising the question of the interaction of deep abbreviation with memory consumption, and Fabrice Parrennes for providing the clause sets mentioned in the paper.

References

1. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Fourth ACM Conference on Computer and Communications Security*. ACM Press, 1997.
2. L. Bachmair and H. Ganzinger. Resolution theorem proving. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 2, pages 19–99. North-Holland, 2001.
3. R. E. Bryant. Graph-based algorithms for boolean functions manipulation. *IEEE Trans. Comp.*, C35(8):677–692, 1986.
4. K. Claessen and N. Sörensson. New techniques that improve MACE-style finite model building. Proc. CADE-19 Workshop W4, July 2003.
5. O. Coudert. *SIAM : Une Boîte à Outils Pour la Preuve Formelle de Systèmes Séquentiels*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, Paris, Oct. 1991.
6. 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. IEEE Computer Society Press, 1991.
7. E. Goto. Monocopy and associative algorithms in an extended Lisp. Technical report, U. Tokyo, 1974.
8. J. Goubault-Larrecq. HimML: Standard ML with fast sets and maps. In *Proc. 5th ACM SIGPLAN Workshop on Standard ML and its Applications (ML'94)*, pages 62–69, Orlando, Florida, USA, 1994. Implementation available on `http://www.lsv.ens-cachan.fr/~goubault/himml-dwnld.html`.
9. J. Goubault-Larrecq. Deciding $\mathcal{H}_1$ by resolution. *Information Processing Letters*, 95(3):401–408, Aug. 2005.
10. J. Goubault-Larrecq. Determinization and models. *Information Processing Letters*, 2006. Submitted.
11. 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), Paris, France, Jan. 2005. Springer-Verlag LNCS 3385.
12. J. N. Hooker. Testing heuristics: We have it all wrong. *Journal of Heuristics*, 1:33–42, 1996.
13. F. Jacquemard, M. Rusinowitch, and L. Vigneron. Tree automata with equality constraints modulo equational theories. Research report, INRIA Futurs, 2005.
14. S. Limet and G. Salzer. Manipulating tree tuple languages by transforming logic programs. *Electronic Notes in Theoretical Computer Science*, 86(1), 2003. Proceedings of the 4th Intl. Workshop on First-Order Theorem Proving (FTP'2003), Valencia, Spain, June 2003. Long version, LIFO Research Report RR-2004-01, March 2004, Orléans, France, available from `http://www.univ-orleans.fr/SCIENCES/LIFO/prodsci/rapports/RR/RR2004/RR-2004-01.ps.gz`.
15. R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.
16. F. Nielson, H. R. Nielson, and H. Seidl. Normalizable Horn clauses, strongly recognizable relations and Spi. In *Proc. 9th Static Analysis Symposium*, pages 20–35. Springer Verlag LNCS 2477, 2002.
17. A. Pettorossi and M. Proietti. Transformation of logic programs. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 697–787. Oxford University Press, 1998.
18. A. Riazanov and A. Voronkov. Splitting without backtracking. In B. Nebel, editor, *Proc. 17th Intl. Joint Conf. Artificial Intelligence*, volume 1, pages 611–617. Morgan Kaufmann, Aug. 2001.
19. A. Riazanov and A. Voronkov. Limited resource strategy in resolution theorem proving. *Journal of Symbolic Computation*, 36(1-2):101–115, July 2003.
20. G. Sutcliffe and C. B. Suttner. The TPTP problem library: CNF release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
21. T. Tammet. Finite model building: Improvements and comparisons. Proc. CADE-19 Workshop W4, July 2003.
22. C. Weidenbach. *Computational Aspects of a First-Order Logic with Sorts*. PhD thesis, Technische Fakultät der Universität des Saarlandes, Saarbrücken, Germany, 1996.
23. C. Weidenbach. Sorted unification and tree automata. In W. Bibel and P. Schmitt, editors, *Applied Logic*, volume 1, chapter 9, pages 291–320. Kluwer, 1998.
24. C. Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In *Proc. 16th Intl. Conf. Automated Deduction*, pages 314–328. Springer-Verlag LNAI 1632, 1999.
25. C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topić. SPASS version 2.0. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction*. Springer-Verlag LNAI 2392, 2002.
26. C. Weidenbach, B. Gaede, and G. Rock. SPASS & FLOTTER version 0.42. In M. A. McRobbie and J. K. Slaney, editors, *Proc. 13th Intl. Conf. Automated Deduction*, pages 141–145. Springer-Verlag LNCS 1104, 1996.

Address for Offprints: ENS Cachan
61 avenue du président-Wilson,
F-94235 Cachan Cedex