Electronic Notes in Theoretical Computer Science 70 No. 2 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume70.html 14 pages

Memoization-based proof search in LF: an experimental evaluation of a prototype

Brigitte Pientka [1]

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA

Abstract

Elf is a general meta-language for the specification and implementation of logical systems in the style of the logical framework LF. Proof search in this framework is based on the operational semantics of logic programming. In this paper, we discuss experiments with a prototype for memoization-based proof search for *Elf* programs. We compare the performance of memoization-based proof search, depth-first search and iterative deepening search using two applications: 1) Bi-directional type-checker with subtyping and intersection types 2) Parsing of formulas into higher-order abstract syntax. These experiments indicate that memoization-based proof search is a practical and overall more efficient alternative to depth-first and iterative deepening search.

1 Introduction

Deductive systems, given via axioms and inference rules, play an important role when formalizing the behavior of programs and providing some guarantee about it. In research on "certifying code", safety properties are expressed as deductive systems and programs are equipped with a certificate (proof) that asserts certain safety properties. Before executing the program, the host machine can then quickly verify the code's safety properties by checking the certificate against the program. Several approaches to certifying code use the logical framework LF [6] as a meta-language for specifying safety policies. Appel and Felty [1] implement their safety policy using the higher-order logic programming language *Elf* [11] which is based on LF. To verify that a given program fulfills a specified safety policy, the specification is executed by the logic programming interpreter. Necula and Rahul [8] use a logic programming interpreter based on a subset of LF for checking the correctness

[1] Email: bp@cs.cmu.edu

of certificates. In their approach, the certificate is a bit-string which guides a logic programming interpreter to resolve non-deterministic choices during proof reconstruction. One major benefit of using a general framework such as LF is that it reduces the effort required for each particular policy. By assigning a logic programming interpretation to LF, we obtain a generic proof search procedure. The higher-order logic programming interpreter for *Elf* uses a depth-first search strategy. This is unsatisfying as many straightforward specifications are not executable at all. Moreover, performance may be unacceptable due to redundant computation. A fair search strategy such as iterative deepening, as used in the meta-theorem-prover *Twelf* [17], makes proof search over left-recursive grammars feasible, but as experiments show, redundant computation also severely hampers performance.

In this paper, we discuss experiments with proof search based on memoization. Some redundant computation is eliminated by memoizing sub-computation and re-using their result later. This technique of memoization, also known as tabling, has been successfully applied for first-order logic programming to solve complex problems such as implementing recognizers and parsers for grammars [20], representing transition systems CCS and writing model checkers [3]. The XSB system [16] demonstrates that tabled logic programs can be executed efficiently and in fact can be mixed with Prolog programs to achieve the best of both worlds. In [15] we give a high-level description of *tabled higher-order logic programming* based on sequent calculi and controlled cuts. Based on it, we have implemented a prototype for evaluating *Elf* programs. In this paper, we compare the performance of proof search based memoization, depth-first search, iterative deepening using two applications: 1) bi-directional type checking using subtyping and intersection types and 2) parsing into higher-order abstract syntax. As the experiments demonstrate proof search based on memoization can lead to substantial performance improvements, making the execution of some queries at all feasible. Although we concentrate here on the logical framework LF, which is the basis of *Elf*, it seems possible to apply the presented approach to λProlog [7] or Isabelle [9], which are based on hereditary Harrop formulas and simply typed terms.

The paper is organized as follows: In Sec. 2 we introduce a small functional language with subtyping and present the idea behind higher-order tabled computation. Challenges faced in the higher-order setting are described in Sec. 3. In Sec. 4 we discuss experiments with an bi-directional type checking algorithm for this functional language and compare the performance of proof search based on memoization with traditional logic programming. In Sec. 5, we discuss experiments with parsing first-order formulas into higher-order abstract syntax and compare proof search based on memoization with proof search based on iterative deepening. In Sec. 6 we discuss related work and summarize the results and outline future work.

2 Illustrating example: subtyping

As a starting point, we introduce a small functional language with subtyping. We also include an example data-type bit for strings of bits along with the subtypes nat for natural numbers (without leading zeroes), pos for positive numbers and zero for 0.

$$\begin{array}{llll} \text{expressions} & e & ::= & \epsilon \mid e\ \mathsf{0} \mid e\ \mathsf{1} \mid \mathsf{lam}\ x.e \mid \mathsf{app}\ e_1\ e_2 \mid \mathsf{let}\ u = e_1\ \mathsf{in}\ e_2 \mid \\ & & & \mathsf{case}\ e\ \mathsf{o}f\ \epsilon\ \Rightarrow e_1 | x\ \mathsf{0} \Rightarrow e_2 | x\ \mathsf{1} \Rightarrow e_3 \\ \text{types} & \tau & ::= & \mid \mathsf{zero} \mid \mathsf{pos} \mid \mathsf{nat} \mid \mathsf{bit} \mid \tau_1 \rightarrow \tau_2 \end{array}$$

We represent 0 and 1 as constructors and ϵ represents the empty string. For example, 6 is represented as ϵ110. This example is small enough that it allows us to illustrate the basic principles and challenges of proof search based on memoization in the setting of LF.

```
refl: sub T T.     zn: sub zero nat.   arr: sub (T1 => T2) (S1 => S2)
tr:   sub T S      pn: sub pos nat.         <- sub S1 T1
      <- sub T R   nb: sub nat bit.         <- sub T2 S2.
      <- sub R S.

tp_sub: of E T      tp_lam: of (lam ([x] E x)) (T1 => T2)
        <- of E T'          <- ({x:exp} of x T1 -> of (E x) T2)}.
        <- sub T' T.
```

For implementing the subtyping relations logic programming based on Horn clauses suffices. However, *Elf* is much richer than first-order logic programming and also supports elegant encodings based on higher-order abstract syntax [14]. Variables bound in constructors such as lam will be bound with λ in *Elf*. The binding described by λ-expression $\lambda x.Ex$ is denoted by `[x] E x` using *Elf* syntax and the Mini-ML expression lam $x.e$ is represented as `lam [x] E x` in *Elf*. Substitution is modeled via application and β-reduction. In addition to the variable binding construct, *Elf* supports reasoning from hypotheses and handling parameters. The premise of the typing rule for lam depends on the new parameter x and the hypothesis that x is of type τ_1. Moreover, we assume that it is possible to rename all variables in e, if necessary. In *Elf* this is represented by `({x:exp} of x T1 -> of (E x) T2)` where `{x:exp}` denotes the universal quantifier $\Pi x\colon \mathrm{exp}$. We can show `of (lam ([x] E x)) (T1 => T2)`, if we can prove that for a new variable `x`, if `x` has type `T1` then the body of the function `(E x)` has type `T2`. For a more detailed discussion see [12] [2]. Type inference in the give system is highly non-deterministic. For example, a logic programming interpreter might get trapped in applying the transitivity rule leading to an infinite loop. In addition, we repeatedly type-check subexpressions, which occur more than once. This might severely hamper the performance.

[2] The code to all examples can be found at `http://www.cs.cmu.edu/~bp/LFM02`

Tabling methods evaluate programs by maintaining tables of subgoals and their answers and by resolving repeated occurrences of subgoals against answers from the table. We review briefly Tamaki and Sato's multi-stage strategy [18], which differs only insignificantly from SLG resolution [2] for programs without negation. To demonstrate tabled computation, we consider the evaluation of the query sub zero T in more detail. The search proceeds in multiple stages. The table serves two purposes: 1) We record all sub-goals encountered during search. If the current goal is not in the table, then we add it to the table and proceed with the computation. Computation at a node is suspended, if the current goal is a variant of a table entry. 2) In addition to the sub-goals we are trying to solve, we also store the results of computation in the table as a list of answers to the sub-goals. To simplify the table in this presentation, we do not record the certificate (proof term) explicitly in the table, although we do record a proof skeleton in the actual implementation. Using the skeleton, we can reconstruct the proof, if desired. This means we can not only detect infinite and redundant paths, but also make progress by re-using the answers from the table. In each stage we apply program clauses and answers from the table. Figure 1 illustrates the search process.

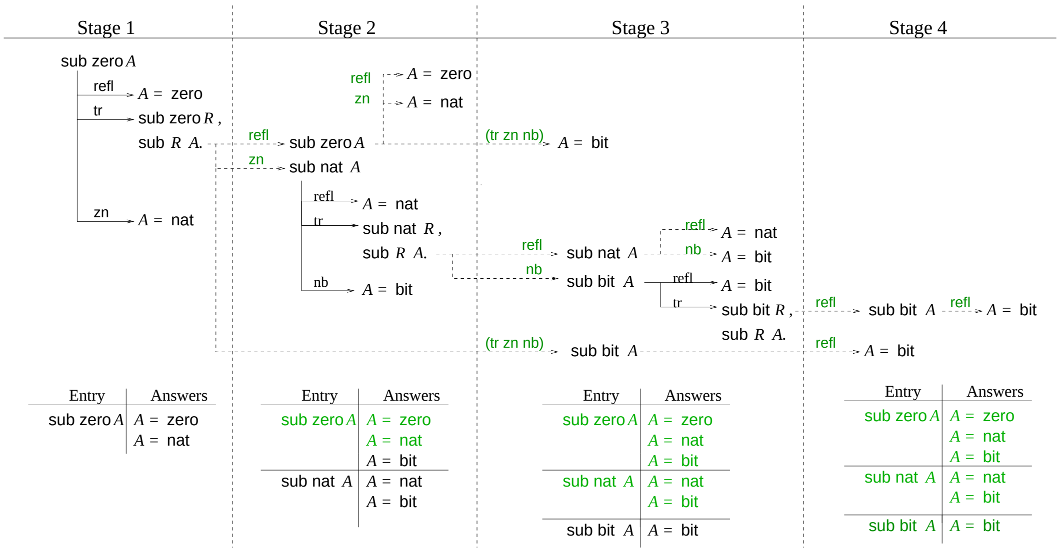

Fig. 1. Staged computation

The root of the search tree is labeled with the goal sub zero A. Each node is labeled with a goal statement and each child node is the result of applying a program clause or an answer from the table to the *leftmost* atom of the parent node. Applying a clause $H \leftarrow A_1 \leftarrow A_2 \ldots \leftarrow A_n$ results in the subgoals $A_1, A_2, \ldots, A_n$ where all of these subgoals need to be satisfied. We will then expand the first subgoal A_1 carrying the rest of the subgoals $A_2, \ldots, A_n$ along. If a branch is successfully solved, we show the obtained answer. To distinguish between program clause resolution and re-using of answers, we have two different kinds of edges in the tree. The edges obtained by program clause

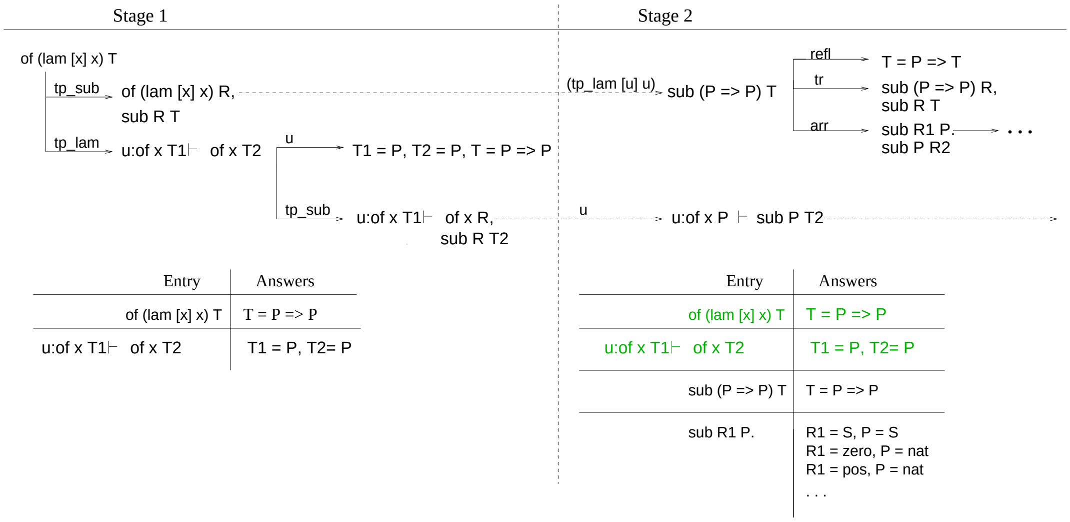

Fig. 2. Staged computation for identity function

resolution are solid while edges obtained by reusing answers from the table are dashed. Both are labeled with the clause name that was used to derive the child node. Using the labels at the edges we can reconstruct the proof term for a given query. In general, we will omit the actual substitution under that the parent node unifies with the program clause to avoid cluttering the example. To ensure we generate all possible answers for the query, we restrict the re-use of answers from the table. In each stage, we are only allowed to re-use answers that were generated in previous stages. Answers from previous stages (available for answer resolution) are marked gray, while current answers (not available yet) are black.

3 Challenges in LF

In tabled higher-order logic programming, we extend tabling to handle subgoals that may contain implication and universal quantification and our term language is the dependently typed λ-calculus. The table entries are no longer atomic goals, but goals A together with a context Γ of assumptions. In addition, terms might depend on assumptions on Γ. To highlight some of the challenges we discuss the evaluation of query $\mathsf{of}\ (\mathsf{lam}\ [x]\ x)\ T$ (see Fig. 2).

The possibility of nested implications and universal quantifiers adds a new degree of complexity to memoization-based computation. Retrieval operations on the table need to be redesigned. One central question is how to look up whether a goal $\Gamma \vdash a$ is already in the table. There are two options: In the first option we only retrieve answers for a goal a given a context Γ, if the goal together with the context matches an entry $\Gamma' \vdash a'$ in the table. In the second option we match the subgoal a against the goal a' of the table entry $\Gamma' \vdash a'$, and treat the assumptions in Γ' as additional subgoals, thereby delaying satisfying these assumptions. We choose the first option of retrieving goals together with their dynamic context Γ'. One reason is that it restricts the number of possible retrievals early on in the search and the possible ways these dynamic assumptions could be satisfied. For example, to solve subgoal $u{:}\mathsf{of}\ x\ T_1 \vdash \mathsf{of}\ x\ R, \mathsf{sub}\ R\ T_2$, we concentrate on solving the left-most goal $u{:}\mathsf{of}\ x\ T_1 \vdash \mathsf{of}\ x\ R$ keeping in mind that we still need to solve $u{:}\mathsf{of}\ x\ T_1 \vdash \mathsf{sub}\ R\ T_2$. as there exists a table entry $u{:}\mathsf{of}\ x\ T_1 \vdash \mathsf{of}\ x\ T_2$, which is a variant of the current goal $u{:}\mathsf{of}\ x\ T_1 \vdash \mathsf{of}\ x\ R$, computation is suspended.

Due to the higher-order setting, the predicates and terms might depend on Γ. Virga [19] developed in his PhD thesis techniques, called subordination checking, to analyze *Elf* programs statically before execution and construct a dependency graph. Using type dependency analysis based on subordination [19] we can detect and eliminate such dependencies. This allows us to detect more loops in the search tree, i.e. more nodes can be suspended because a variant of it is already in the table. Another optimization concerns the handling of assumptions in context Γ. When storing a goal A together with a context Γ, not all assumptions in the dynamic context Γ might contribute to the proof of goal G. For example, during the second stage of evaluating the query above we derive the following two subgoals: $\mathsf{of}\ x\ T_1 \vdash \mathsf{sub}\ R\ T_2$ and $\mathsf{sub}\ R\ T_2$. Any solution to the goal $\mathsf{sub}\ R\ T_2$ will be independent of the assumption $\mathsf{of}\ x\ T_1$. We can again use type dependency analysis based on subordination to strengthen the context Γ leading to smaller tables and the elimination of more redundant and infinite paths.

While computation using memoization yields better performance for programs with transitive closure or left-recursion, Prolog-style evaluation is more efficient for right recursion. For example, Prolog has linear complexity for a simple right recursive grammar, but with memoization the evaluation could be quadratic as calls need to be recorded in the tables using explicit copying. Therefore it is important to allow tabled and non-tabled predicates to be freely intermixed and be able to choose the strategy that is most efficient for the situation at hand. Hence in the current prototype, the user declares predicates to be tabled, if he/she wishes to use memoization for it. Mixing tabled and non-tabled predicates is essential, in order to obtain an efficient proof search engine.

4 Example: Refinement type checking

In this section, we discuss experiments with a bi-directional type-checking algorithm for a small functional language with intersection types which has been developed by Davies and Pfenning [4]. Type inference in a functional language with subtyping and intersection types is usually considered impractical, as no principal types exist. The idea behind bi-directional type-checking is to distinguish expressions for which a type can be synthesized from expressions which can be checked against a given type. The programmer specifies some types to guide inferring a type for certain expressions.

$$
\begin{array}{llll}
\text{Inferable} & I & ::= & x \mid \epsilon \mid I\ \mathsf{0} \mid I\ \mathsf{1} \mid \mathsf{app}\ IC \mid C : \tau \\
\text{Checkable} & C & ::= & I \mid \mathsf{lam}\ x.C \mid \mathsf{let}\ u = I\ \mathsf{in}\ C \mid \\
 & & & \mathsf{case}\ I\ \mathsf{o}f\ \epsilon\ \Rightarrow C_1 | x\ \mathsf{0} \Rightarrow C_2 | x\ \mathsf{1} \Rightarrow C_3
\end{array}
$$

The intention is that given a context Γ and an expression I, we use type-inference to show expression I has type τ and type-checking for verifying that expression C has type τ. In an implementation of the bi-directional type checking algorithm, there may be many ways to derive that I has a type τ and similarly there are more than one way to check that C has a given type τ. To discuss the full bi-directional type-checking algorithm is beyond the scope of this paper and the interested reader is referred to the original paper by Davies and Pfenning [4].

We use an implementation of the bi-directional type-checker in *Elf* by Pfenning. The type-checker is executable with the original logic programming interpreter, which performs a depth-first search. However, redundant computation may severely hamper its performance as there are several derivations for proving that a program has a specified type. For example, there are 20,000 ways to show that the program plus has the intersection type $(\mathsf{nat} \rightarrow \mathsf{nat} \rightarrow \mathsf{nat}) \wedge (\mathsf{nat} \rightarrow \mathsf{pos} \rightarrow \mathsf{pos}) \wedge (\mathsf{pos} \rightarrow \mathsf{nat} \rightarrow \mathsf{pos}) \wedge (\mathsf{pos} \rightarrow \mathsf{pos} \rightarrow \mathsf{pos})$. It might be argued that we are only interested in one proof for showing that plus has the specified type and not in all of them. However, it indicates that already fairly small programs involve a lot of redundant computation. More importantly than succeeding quickly may be to fail quickly, if plus has not the specified type. Failing quickly is essential in debugging programs and response times of several minutes are unacceptable. In the experiments, we are measuring the time it takes to explore the whole proof tree. This gives us an indication how much redundancy is in the search space. When checking a term C against a type τ, we use memoization. The proof search based on memoization uses strengthening and variant checking. We have tested programs for addition, subtraction and multiplication. Note that in order to for example type-check the multiplication program, we need to also type-check any auxiliary programs used such as addition and shift. The number associated to each program name denotes the depth of the intersection type associated to it. For example, plus4 means we assigned 4 types by using intersections to plus. mult1a indicates we associated one possible type to multiplication program. If a program label is marked with (np), it means this query does not have a solution and the type-checker should reject the query.

For type-checking programs plus and multiplication proofs search based on memoization outperforms depth-first search. This is surprising, as currently no sophisticated indexing is used for accessing the table. It indicates that already simple memoization mechanism can substantially improve performance. In the case of multiplication, it makes type-checking possible at all. We stopped the depth-first search procedure after 10h. Of course proof search

Program	Depth-First	Memoization	#Entries	#SuspGoals
plus'4	483.070 sec	2.330 sec	151	48
plus4	696.730 sec	3.150 sec	171	74
plus4(np)	22.770 sec	1.95 sec	143	56
sub'1a	0.070 sec	0.240 sec	58	11
sub'3a	0.130 sec	0.490 sec	92	20
sub1a	3.52 sec	7.430 sec	251	135
sub1b	3.88 sec	7.560 sec	252	138
sub3a	10.950 sec	9.970 sec	277	167
sub3b	10.440 sec	11.200 sec	278	170
mult1(np)	1133.490 sec	4.690 sec	217	83
mult1a	807.730 sec	4.730 sec	211	78
mult1b	2315.690 sec	6.050 sec	226	101
mult1c	2963.370 sec	5.310 sec	226	107
mult4	∞	17.900 sec	298	270
mult4(np)	∞	13.140 sec	275	194

Table 1
Finding all solutions: depth-first vs. memoization-based search

based on memoization has some overhead in storing and accessing goals in the table. As indicated with subtraction programs, this overhead might degrade the performance of memoization based proof search. When type-checking subtraction program depth-first search performs better than memoization-based search for the first 5 sample programs. In the last example sub3b however memoization-based search wins over depth-first search.

For fairness, we include also a comparison between the two search strategies, when we stop after the first answer has been found. It is apparent that currently finding the first solution to a solvable type-checking problem (i.e. it is provable that the program has the specified type) always takes longer with proof search based on memoization – in some cases considerably longer (see subtraction and multiplication). This is an indication that accessing the table is still quite expensive in the current implementation. This comes as no real surprise, because no indexing has been implemented so far. The other reason memoization-based search does fairly badly is due the multi-stage search strategy. Although this strategy is relatively easy to implement and understand, it restricts retrieval of answers present in the table to answers generated

Program	Depth-First	Memoization	#Entries	#SuspGoals
plus'4	0.08 sec	0.180 sec	54	0
plus4	0.1 sec	0.430 sec	72	0
plus4(np)	22.770 sec	1.95 sec	143	56
sub'1a	0.050 sec	0.240 sec	64	11
sub'3a	0.110 sec	0.410 sec	92	20
sub1a	0.250 sec	6.210 sec	251	135
sub1b	0.250 sec	5.020 sec	242	121
sub3a	0.280 sec	7.80 sec	277	161
sub3b	0.350 sec	8.160 sec	278	164
mult1(np)	1133.490 sec	4.690 sec	217	83
mult1a	0.160 sec	2.900 sec	201	60
mult1b	0.180 sec	4.090 sec	222	90
mult1c	0.170 sec	2.930 sec	211	60
mult4	0.250 sec	7.150 sec	272	181
mult4(np)	∞	13.020 sec	275	194

Table 2
Finding the first solution: depth-first vs. memoization-based search

in previous stages. This causes subgoals to be suspended, although answers might be available, and solving those subgoals is delayed. For this reason, XSB uses SCC (strongly connected component) scheduling strategy, which allows to re-use answers from the table as soon as they are available. But the benefits of memoization-based search are apparent when comparing the time it takes to reject a program by the type-checker as the examples plus4(np) and mult1(np) indicate. Overall, the performance of the memoization based search is much more consistent, i.e. it takes approximately the same time to accept or reject a program.

To make bi-directional type-checking reasonably efficient in practice, Davies and Pfenning currently investigate an algorithm which synthesizes all types of an inferable term and tracks applicable ones through the use of boolean constraints. This is however far from trivial and refining the *Elf* implementation by adding explicit support for memoization, complicates the type checker. As a consequence, the certificates, which are produced as a result of the execution, are larger and contain references to the explicit memoization data-structure. This is especially undesirable in the context of certified code where certificates are transmitted to and checked by a consumer, as sending larger certificates takes up more bandwidth and checking them takes more time. Moreover, proving the correctness of the type-checker with special memoization support will be hard, because we need to reason explicitly about the structure of memoization. The experiments demonstrate that proof search based on memoization has the potential to turn the bi-directional type-checking algorithm into a quite efficient type-checker without any extra effort on behalf of the user.

5 Example: Parser for formulas

Recognition algorithms and parsers for grammars are an excellent way to illustrate the benefits of tabled evaluation. Warren [20] notes that implementations of context-free grammars in Prolog result in a recursive descent recognizer, while tabled logic programming turns the same grammar into a variant of Early's algorithm (also known as active chart recognition algorithm) whose complexity is polynomial. Moreover, tabled logic programming allows us to execute left and right recursive grammars that would otherwise loop under Prolog-style execution (e.g. left recursive ones). We illustrate tabled computation with parsing of first-order formulas into higher-order abstract syntax. First-order formulas are defined as usual.

$$\text{Propositions} \quad A \quad ::= \quad \mathsf{atom}\ P \mid \neg A \mid A\ \&\ A \mid A \vee A \mid A \Rightarrow A \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{forall}\ x.A \mid \mathsf{exists}\ x.A \mid (A)$$

Terms are either constants or variables or functions with arguments. Atomic propositions are either propositional constant or a predicate with terms as arguments. In addition to the given grammar, we impose the following precedence ordering: $\neg > \& > \vee > \Rightarrow$. Conjunction and disjunction are left associative, while implication is right associative.

```
fall: fq C ('forall' ; I ; F) F' (forall P)}
      <- ({x:id} fq ((bvar I x) # C) F F' (P x)).
fex:  fq C ('exist' ; I ; F) F' (exist P)}
      <- ({x:id} fq ((bvar I x) # C) F F' (P x)).
cq :  fq C F F'
      <- fi C F F' P.
% implication -- right associative
fimp: fi C F F' (P1 => P2)
      <- fo C F ('imp' ; F1) P1
      <- fi C F1 F' P2.
ci:   fi C F F' P
      <- fo C F F' P.
% disjunction -- left associative
for: fo C F F' (P1 v P2)
     <- fo C F ('or' ; F1) P1
     <- fa C F1 F' P2.
```

The parser takes a context for the bound variables, two lists of tokens and returns a valid formula represented in higher-order abstract syntax [14]. Initially the context of bound variables is empty, the first list of tokens represents the input stream, the second list is empty and P will eventually contain the result. Using higher-order abstract syntax, variables bound in constructors such as forall and exists will be bound with λ in *Elf*. The simplest way to implement left and right associativity properties of implications, conjunction and disjunction is to mix right and left recursive program clauses. Clauses for conjunction and disjunction are left recursive, while the program clause for implication is right recursive.

Such an implementation of the grammar is straightforward mirroring the defined properties such as left and right associativity and precedence ordering. However, the execution of the grammar will loop infinitely when executed with a traditional logic programming interpreter. Hence we compare execution of some sample programs using proof search based on memoization and with proof search based on iterative deepening, as performed by the current theorem prover *Twelf* [17]. Iterative deepening search requires the user to provide a depth-bound. It is worth pointing out that iterative deepening search will stop after it found its first solution or it hits a depth-bound, while search based on memoization will stop after it found a solution (and showed that no other solution exists) or proved no other solution exists. This means, we cannot use iterative deepening search to decide whether a given stream of tokens should be accepted or not, while the memoization-based search yields a decision procedure.

Length of input	Iter. deepening	Memoization	#Entries	#SuspGoals
5	0.020 sec	0.010 sec	15	11
20	1.610 sec	0.260 sec	60	54
32	208.010 sec	2.020 sec	176	197
56	∞	7.980 sec	371	439
107	∞	86.320 sec	929	1185

Table 3
Comparison between iterative deepening and memoization based search

The experiments indicate that proof search based on memoization provides a more efficient way to decide whether a given stream of tokens belongs to the language of formulas. In fact for input streams whose length is greater than 50 tokens, we stopped the iterative deepening procedure after several hours.

Remark 1: It is worth noting that in general the straightforward approach of adding an extra argument to the non-terminals of the input grammar – representing the portion of the parse tree that each rule generates – and naturally to also add the necessary code that constructs the parse tree can be extremely unsatisfactory from a complexity standpoint. Polynomial recognition algorithms might be turned into exponential algorithm since there may be exponentially many parse trees for a given input string. However in this example, there is exactly one parse trace associated to each formula, therefore adding an extra argument to the recognizer only adds a constant factor.

Remark 2: Instead of representing the input string as a list, we can store it in the database as a set of facts. We can think of each token in the input stream being numbered starting from 1. Then we will store the string as a set of facts of the form `word 1 'forall'.` `word 2 'x'.` etc. where `word i tok_i` represents the ith token of the input stream.

6 Related Work and Conclusion

A number of different frameworks similar to *Elf* have been proposed such as λProlog [7,5] or Isabelle [9,10]. While *Elf* is based on the LF type theory, λProlog and Isabelle are based on hereditary Harrop formulas. The traditional approach for supporting theorem proving in these frameworks is to guide proof search using tactics and tacticals. Tactics transform a proof structure with some unproven leaves into another. Tacticals combine tactics to perform more complex steps in the proof. Tactics and tacticals are written in ML or some other strategy language. To reason efficiently about some specification, the user implements specific tactics to guide the search. This means that tactics have to be rewritten for different specifications. Moreover, the user has to understand how to guide the prover to find the proof, which often requires expert knowledge about the systems. Proving the correctness of the tactic is itself a complex theorem proving problem. The approach taken in *Elf* is to endow the framework with the operational semantics of logic programming and design general proof search strategies for it. The user can concentrate on developing the high-level specification rather than getting the proof search to work. The correctness of the implementation is enforced by type-checking alone. The preliminary experiments demonstrate that proof search based on memoization offers a powerful search engine.

Proof search based on memoization will only find one representative proof for a given query, and for this reason it is inherently incomplete. For example, when type-checking $\mathsf{plus}\ \ (\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}) \wedge (\mathsf{nat} \to \mathsf{pos} \to \mathsf{pos}) \wedge (\mathsf{pos} \to \mathsf{nat} \to \mathsf{pos}) \wedge (\mathsf{pos} \to \mathsf{pos} \to \mathsf{pos})$ has many proofs under the traditional logic programming interpretation, but we will find only one with memoization-based search. However, we often do not want and need to distinguish between different proofs for a formula A, but only care about the existence of a proof for A together with a proof term. In [13] Pfenning develops a dependent type theory for proof irrelevance and discusses potential applications in the logical framework. This allows us to treat all proofs for A as equal where two proofs are considered equal if they produce the same answer substitution. In the future we plan to combine the efforts on proof search based on memoization and proof irrelevance into one framework.

From the experience gathered with the prototype, it seems that also other logic programming languages such as λProlog would profit from memoization-based search. While we have taken care in the design of the prototype, there is much room for improvements. The most pressing issue currently is to implement indexing for the table to enable fast table access.

Acknowledgment: The author gratefully acknowledges numerous discussion with Frank Pfenning and David S.Warren concerning this work. Thanks also to many useful comments from Carsten Schürmann, Bob Harper and Kevin Watkins. This work was partially supported by NSF Grant CCR-9988281.

References

[1] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*, pages 243–253, Jan. 2000.

[2] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[3] B. Cui, Y. Dong, X. Du, K. N. Kumar, C.R. Ramakrishnan, I.V. Ramakrishnan, A. Roychoudhury, S.A. Smolka, and D.S. Warren. Logic programming and model checking. In *Principles of Declarative Programming (Proceedings of PLILP/ALP'98)*, volume 1490 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 1998.

[4] Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *Proceedings of the International Conference on Functional Programming (ICFP 2000), Montreal, Canada*, pages 198–208. ACM Press, 2000.

[5] Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, August 1993.

[6] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[7] Gopalan Nadathur and Dale Miller. An overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.

[8] G. Necula and S. Rahul. Oracle-based checking of untrusted software. In *28th ACM Symposium on Principles of Programming Languages (POPL01)*, 2001.

[9] Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[10] Lawrence C. Paulson. Isabelle: The next seven hundred theorem provers. In *Proceedings of the 9th International Conference on Automated Deduction*, pages 772–773, Argonne, Illinois, 1988. Springer Verlag LNCS 310. System abstract.

[11] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[12] Frank Pfenning. *Computation and Deduction.* Cambridge University Press, 2000. In preparation. Draft from April 1997 available electronically.

[13] Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *16th Annual IEEE Symposium on Logic in Computer Science*, Boston, USA, 2001.

[14] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.

[15] Brigitte Pientka. A proof-theoretic foundation for tabled higher-order logic programs. In *18th International Conference on Logic Programming, Copenhagen, Denmark*, LNCS, page to appear. Springer-Verlag, 2002.

[16] Konstantinos Sagonas and Terrance Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586–634, 1998.

[17] Carsten Schürmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for LF. In *Proceedings of the 15th International Conference on Automated Deduction (CADE-15)*, pages 286–300, Lindau, Germany, July 1998. Springer-Verlag LNCS 1421.

[18] H. Tamaki and T. Sato. OLD resolution with tabulation. In *Proceedings of the 3rd International Conference on Logic Programming*, volume 225 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 1986.

[19] Roberto Virga. *Higher-Order Rewriting with Dependent Types.* PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, 2000.

[20] David S. Warren. *Programming in tabled logic programming.* draft available from http://www.cs.sunysb.edu/˜warren/xsbbook/book.html, 1999.