Tabled higher-order logic programming

Brigitte Pientka

Carnegie Mellon University

Abstract. *Elf* is a general meta-language for the specification and implementation of logical systems in the style of the logical framework LF. Based on a logic programming interpretation, it supports executing logical systems and reasoning with and about them, thereby reducing the effort required for each particular logical system. The traditional logic programming paradigm is extended by replacing first-order terms with dependently typed λ-terms and allowing implication and universal quantification in the bodies of clauses. These higher-order features allow us to model concisely and elegantly conditions on variables and the discharge of assumptions, which are prevalent in many logical systems. However, many specifications are not executable under the traditional logic programming semantics and performance may be hampered by redundant computation.
To address these problems, I propose a tabled higher-order logic programming interpretation for *Elf*. Some redundant computation is eliminated by memoizing sub-computation and re-using its result later. If we do not distinguish different proofs for a property, then search based on tabled logic programming is complete and terminates for programs with bounded recursion. In this proposal, I present a proof-theoretical characterization for tabled higher-order logic programming. It is the basis of the implemented prototype for tabled logic programming interpreter for *Elf*. Preliminary experiments indicate that many more logical specifications are executable under the tabled semantics. In addition, tabled computation leads to more efficient execution of programs.
The goal of the thesis is to demonstrate that tabled logic programming allows us to automate efficiently reasoning with and about logical systems in the logical framework LF.. To achieve this I intend to show soundness of the search based on tabled logic programming, develop efficient implementation techniques, and demonstrate that this allows many more specifications to be executed and properties proven.

Thesis Committee: Frank Pfenning, Carnegie Mellon University
Robert Harper, Carnegie Mellon University
Dana S. Scott, Carnegie Mellon University
David S. Warren, University of New York at Stony Brook

1 Introduction

One of the challenges in computer science today is to provide some form of guarantee about the run-time behavior of programs. This problem is addressed by research on "certified code" where 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. It has been shown that a wide range of safety policies, based on type systems [24] and first-order logic [2, 27], can be checked efficiently using theorem provers tailored to the specific safety policy. Changing and extending the safety policy requires modifications of the theorem prover. Moreover, every time we want to experiment with a new safety policy, we need to write a new theorem prover, which is not a trivial task.

The logical framework LF [18] is a general meta-language for the specification and implementation of logical systems. Based on the logical framework LF, Pfenning [30] developed a higher-order logic programming language, *Elf*. The traditional logic programming paradigm is extended by replacing first-order terms with dependently typed λ-terms and allowing implication and universal quantification in the bodies of clauses. These higher-order features allow us to model concisely and elegantly conditions on variables and the discharge of assumptions, which are prevalent in many logical systems. This stands in sharp contrast to higher-order features supported in many traditional logic programming languages (see for example [7]) where we can encapsulate predicate expressions within terms to later retrieve and invoke such stored predicates.

The inference rules describing the safety policy are represented as a higher-order logic program. To verify that a given program fulfills a specified safety policy, the specification is executed by a logic programming interpreter. The interpreter does not only generate an answer substitution for the existentially quantified variables, but also a certificate for the actual proof, a proof term. *Elf* offers one generic environment for specifying safety policies and executing them, thereby factoring the effort required to built a prover for a specific safety policy. In addition to checking whether a given program fulfills a specific safety policy, it is equally important to verify properties of the safety policy, for example its soundness. This is especially important if we change and extend the policies. Pfenning and Schürmann [35, 44] demonstrated that it is feasible to automate inductive reasoning about logical specifications and complemented the higher-logic programming interpreter with the meta-theorem prover *Twelf*. A key component of the meta-theorem prover is to derive a goal by applying program clauses, lemmas, assumptions and induction hypotheses. Both the logic programming interpreter and the meta-theorem prover are based on the traditional logic programming semantics, although the actual search strategy employed differs.

However, many specifications are not executable under the traditional logic programming semantics and the performance of implementations may be hampered by redundant computation. The source of these problems lies in the fact that we have three different kinds of computation paths: paths that lead to success, paths that lead to a failure, and infinite paths that do not terminate. *Infinite paths* are clearly undesirable as they do not produce any answer and waste valuable computing resources without making any progress. In addition, the performance of traditional logic programming often suffers from *redundant paths* of computation. If the same sub-goal occurs multiple times in the search tree, then it needs to be proven more than once. In many examples the same answers are generated infinitely many times and computation does not terminate, even if our domain is finite for example.

Tamaki and Sato [47] proposed an interpretation for logic programming based on memoization. Some redundant computation is eliminated by memoizing sub-computation and re-using their result later. The evaluation strategy is complete and terminates for programs that have finitely many answers. In addition, computation is more robust and less sensitive to clause and subgoal reordering. This technique of memoization is also known as tabling, caching, loop detection or fixed-point computation and has been widely investigated in the context of first-order logic programming. The XSB system [43], the most powerful system for tabled logic programming, 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.

Based on this idea, I propose a proof-theoretical characterization for tabled higher-order logic programming. It forms the basis of a prototype for evaluating *Elf* programs that memoizes results of sub-computations and reuses its results later. Preliminary experiments indicate that many more logical specifications are executable under the memoization-based semantics. Examples include the lambda calculus, type systems for subtyping, and polymorphism, refinement types, and graph transition systems. In addition, tabled computation may lead to more efficient execution of programs. As computation based on memoization is more robust, it also seems promising for efficiently automating reasoning *about* logical specifications. Based on this work, I propose to show soundness of the abstract machine that uses memoization, develop efficient implementation techniques, and demonstrate that this allows many more specifications to be executed and properties proven. *The goal of the thesis is to demonstrate that tabled logic programming allows us to efficiently automate reasoning with and about logical systems in the logical framework LF.*

2 A motivating example: subtyping

2.1 Background

The logical framework is ideally suited for the specification of type systems and the implementation of type checkers. As a running example throughout this paper, we will consider a type system for a restricted functional language Mini-ML, which includes subtyping. For now, we only consider a small set of expressions, natural numbers z and $\mathsf{s}(e)$, negative numbers $\mathsf{n}(e)$, functions $\mathsf{lam}\ x.e$ and application $\mathsf{app}\ e_1\ e_2$. The type int denotes all numbers, positive and negative numbers including zero. The type of natural numbers nat is a subtype of the type int . The types zero contains only the number 0, the type pos represents all positive natural number and the type negdescribes the negative numbers. The types zero and pos constitute all natural numbers of type nat.

$$\begin{aligned} e &::= \mathsf{n}(e) \mid \mathsf{z} \mid \mathsf{s}(e) \mid \mathsf{lam}\ x.e \mid \mathsf{app}\ e_1\ e_2 \mid \mathsf{letn}\ u = e_1\ \mathsf{in}\ e_2 \\ \tau &::= \mathsf{neg} \mid \mathsf{zero} \mid \mathsf{pos} \mid \mathsf{nat} \mid \mathsf{int} \mid \tau_1 \rightarrow \tau_2 \end{aligned}$$

Typing rules for Mini-ML expressions and a straightforward specification of the subtyping relation using reflexivity and transitivity can be found in Figure 1. In addition to the usual typing rules tp_negz, tp_neg, tp_zn, tp_zz, tp_sn,tp_sp, tp_lam, tp_app and tp_letn we have a subtyping rule tp_sub, which allows us to infer that an expression e has type τ, if e has type τ' and τ' is a subtype of τ.

$$\frac{}{\Gamma \vdash \mathsf{z} : \mathsf{zero}}\,\mathsf{tp\_zz} \qquad \frac{}{\Gamma \vdash \mathsf{z} : \mathsf{nat}}\,\mathsf{tp\_zn} \qquad \frac{\Gamma \vdash e : \mathsf{nat}}{\Gamma \vdash \mathsf{s}(e) : \mathsf{pos}}\,\mathsf{tp\_sp} \qquad \frac{\Gamma \vdash e : \mathsf{nat}}{\Gamma \vdash \mathsf{s}(e) : \mathsf{nat}}\,\mathsf{tp\_sn}$$

$$\frac{}{\Gamma \vdash \mathsf{n}(\mathsf{z}) : \mathsf{neg}}\,\mathsf{tp\_negz} \qquad \frac{\Gamma \vdash e : \mathsf{neg}}{\Gamma \vdash \mathsf{n}(e) : \mathsf{neg}}\,\mathsf{tp\_neg}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathsf{lam}\ x.e : \tau_1 \to \tau_2}\,\mathsf{tp\_lam} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\mathsf{app}\ e_1\ e_2) : \tau}\,\mathsf{tp\_app}$$

$$\frac{\Gamma \vdash e : \tau' \qquad \tau' \preceq \tau}{\Gamma \vdash e : \tau}\,\mathsf{tp\_sub} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash [e_1/u]e_2 : \tau}{\Gamma \vdash \mathsf{letn}\ u = e_1\ \mathsf{in}\ e_2 : \tau}\,\mathsf{tp\_letn}$$

$$\frac{}{T \preceq T}\,\mathsf{refl} \qquad \frac{T \preceq R \qquad R \preceq S}{T \preceq S}\,\mathsf{tr} \qquad \frac{S_1 \preceq T_1 \qquad T_2 \preceq S_2}{(T_1 \to T_2) \preceq (S_1 \to S_2)}\,\mathsf{arrow}$$

$$\frac{}{\mathsf{zero} \preceq \mathsf{nat}}\,\mathsf{zn} \qquad \frac{}{\mathsf{pos} \preceq \mathsf{nat}}\,\mathsf{pn} \qquad \frac{}{\mathsf{nat} \preceq \mathsf{int}}\,\mathsf{nati} \qquad \frac{}{\mathsf{neg} \preceq \mathsf{int}}\,\mathsf{negi}$$

Fig. 1. Typing rules including subtyping relation

The subtyping relation can be directly translated into *Elf* using logic programming style notation. From a computational point of view, the clause describing transitivity of subtyping is interpreted as follows: To show `sub T S` we need to prove the two subgoals `sub T R` and `sub R S`. From a logical point of view, this clause represents the implication `sub R S -> (sub T R -> sub T S)`, i.e. the subgoals `sub R S` and `sub T R` imply `sub T S`. The axioms zn, pn, nati and negi are represented as atomic clauses. Constants `nat`, `post`, `neg` and `int` represent the basic types and the function type is denoted by `T1 => T2`. Throughout this example, we reverse the arrow $A_1 \to A_2$ writing instead $A_2 \leftarrow A_1$. The complete *Elf* program is given in the appendix.

```
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    nati : sub nat int.          <- sub T2 S2
       <- sub R S.   negi : sub neg int.
```

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 [34]. The key concept of higher-order abstract syntax is to represent variables in Mini-ML by variables in *Elf*. 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*. In addition to the variable binding construct, *Elf* supports reasoning from hypothesis and handling parameters. The premise of typing rule for lam depends on the new parameter x and the hypothesis that x is of type τ_1. Under the assumption that x is a new parameter and x has type τ_1, we can check that e has type τ_2. In *Elf* this is represented by `({x:exp} of x T1 -> of (E x) T2)`.

```
tp_zz  : of z zero.       tp_lam  : of (lam ([x] E x)) (T1 => T2)
tp_zn  : of z nat.                  <- ({x:exp} of x T1 -> of (E x) T2).

tp_sp  : of (s E) pos     tp_app  : of (app E1 E2) T
         <- of E nat.               <- of E1 (T2 => T)
tp_sn  : of (s E) nat               <- of E2 T2.
         <- of E nat.
tp_sub : of E T           tp_letn : of (letn E1 [x] E2 x)) T
         <- of E T'                 <- of E1 T1
         <- sub T' T.               <- of (E2 E1) T.
```

From an computational point of view, we can show `of (lam ([x] E x)) (T1 => T2)`, if we can prove that for any variable `x`, if `x` has type `T1` then the body of the function `(E x)` has type `T2`. From a logical point of view the clause `tp_lam` is interpreted as a nested implication where we quantify over the variable `x`. If for any `x`, `of x T1` implies `of (E x) T2`, then `of (lam ([x] E x)) (T1 => T2)` is true. For a more detailed discussion of the representation of Mini-ML and its type system see [32].

2.2 Traditional logic programming semantics

To illustrate the problems with the traditional logic programming interpretation, let us consider what happens, if we try to compute all supertypes of zero . When executing the query `sub zero T` in *Elf*, we construct not only an answer substitution for T during search, but also a certificate of the actual proof in form of a proof term. Figure 2 shows the search tree for the query `sub zero T`. Each node is labeled with a goal statement and each child node is the result of applying a program clause 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. $\top$ indicates the branch is successfully solved. Each edge is labelled 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. We will omit the actual substitution under that the parent node unifies with the program clause to avoid cluttering the example.

The search tree in Figure 2 illustrates two main problems with the traditional logic programming semantics: *infinite* and *redundant paths of computation*. A depth-first interpreter for example gets trapped in the transitivity rule and it misses answers that are

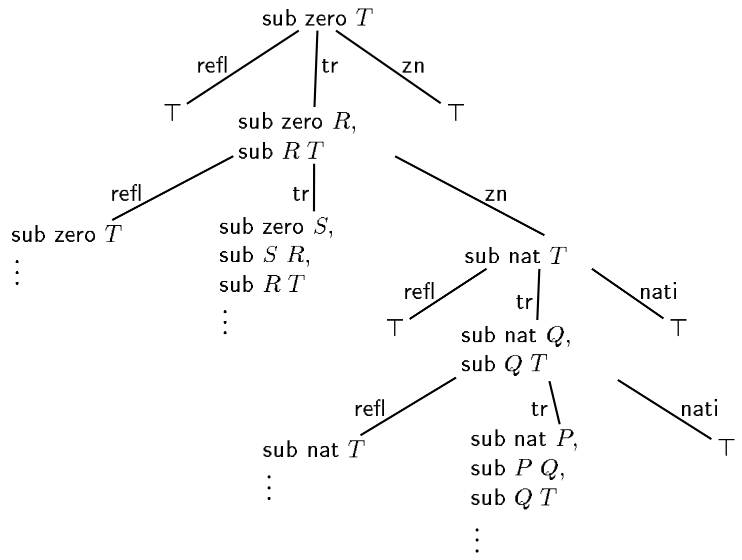


Fig. 2. Search tree

derived in other branches. A breadth-first strategy might seem a sufficient theoretical answer, since it will eventually find any successful computation paths in the search tree, but is practically infeasible and computation may not terminate even if the set of all answers is finite as in the given example. An iterative-deepening strategy tries to combine both strategies, but exhibits essentially the same problems as breadth-first search, i.e. computation will not terminate.

Another observation when studying the search tree is that we repeatedly evaluate the same subgoals. In the search tree in Figure 2 for example, we repeatedly evaluate the subgoal sub zero T or variants of it. This can lead to unacceptable performance and complexity of the search tree especially in the presence of reflexive, symmetric, and transitive relations. Even in the absence of these relations, this problem arises. Let us consider for example type-checking again. To type check a term letn $u = e_1$ in e_2, we first type-check the expression e_1 and then type-check the expression $[e_1/u]e_2$. This means we will repeatedly type-check every occurrence of e_1 in expression e_2. In the expression letn $u =$ lam $x.x$ in (app (lam $y.y$) u), we will type-check lam $x.x$ (or variants of it) three times.

Although this example is small, it demonstrates that when executing the type-checker, we repeatedly type-check the same subgoals leading to redundancy in computation. To eliminate this redundancy, some sophisticated type and sort checkers memoize the result of sub-computations to obtain more efficient type and sort checkers.

In this section we have briefly described the traditional logic programming semantics that forms the basis for executing logic programs and reasoning with and about them. Note that the search tree contains infinite and redundant paths. Problems due to infinite and redundant paths affect the search for a proof in such a tree, independently of whether a depth-first, breadth-first or iterative deepening search strategy is used. Therefore an interpretation that eliminates redundant and infinite paths from the search tree has the potential to improve the execution of logic programs and the reasoning with and about them.

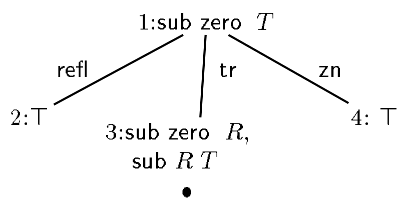
2.3 Tabled logic programming interpretation

To eliminate redundant computation, Tamaki and Sato [47] proposed a new interpretation of logic programs. The central idea is to memoize sub-computation in a table and reuse its result later. The table serves two purposes: 1) We record all sub-goals encountered during search. By checking whether a sub-goal is already in the table, we can detect redundant or infinite paths in the search tree. If the current goal is not in the table, then we add it to the table and proceed with the computation. 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. This means we can not only detect infinite and redundant paths, but also make progress by re-using the answers from the table.

The interaction between storing subgoals and retrieving answers is critical, if we want to obtain a complete search strategy that finds all possible answers to a query. To control the retrieval of answers from the table, we associate a forward pointer with the answer list. The pointer n indicates that all answers 1 to n can be reused. We will come back to this problem at the end of this section.

To demonstrate computation based on memoization, we reconsider the evaluation of the query sub zero T. We start with the root of the search tree, labeled with the goal sub zero T. For easy reference of nodes, we assign to every node a number representing the order of creation. The root node is labeled with number 1, as it is the first node created.

In addition to the search tree, we have a table in which we record all goals, its corresponding answer substitutions and a forward pointer for retrieving answer substitutions. Initially, we are not allowed to use any answers, therefore the forward pointer is 0.



Entry	Answer	Ptr
sub zero T	[zero $/T$]	0
	[nat $/T$]	

Fig. 3. Search tree

We start by solving the goal sub zero T (see Figure 3). Before applying any program clauses, we store the goal sub zero T in the table. Expansion by clause refl gives us the first answer substitution [zero $/T$] to the original query. This answer is recorded in the solution list to sub zero T. When we expand the root node using the clause tr, we obtain a new node 3 with the subgoals sub zero R, sub R T. Since the subgoal sub zero R is an instance of the goal sub zero T in the table, expansion of the node 3 is suspended. By applying clause zn to the root node, we yield a new answer substitution [nat $/T$] to the original query, which is inserted into the solution list. As all possible clauses have been applied, the first stage of computation is completed. At this point, all leaves in the search tree are either success nodes or failure nodes where failure nodes includes the suspended nodes. Before starting the next stage, we update the forward pointer associated to goal sub zero T to point to the end of the answer list. Then we awaken the suspended goals and use the answers 1 and 2 from the answer list to expand node 3 further. This is indicated by dashed lines in the search tree in Figure 4. The new subgoal sub nat T is added to the table (see Figure 5) and program clauses are applied to it. Using clause nati on node 6, we derive a new answer substitution for the original goal [int $/T$].

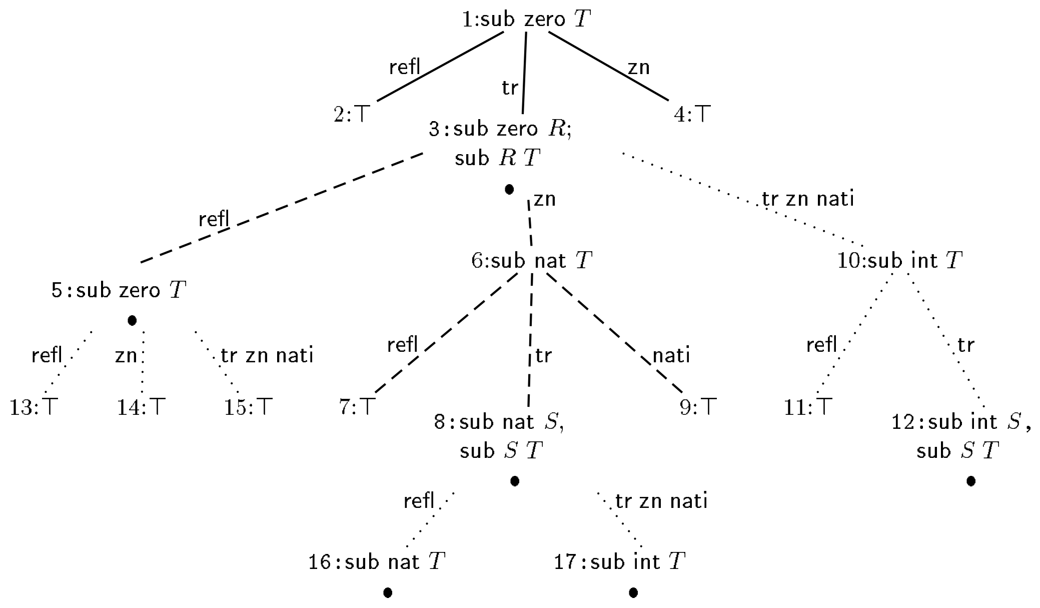


Fig. 4. Search tree in stage 2 and stage 3

In stage 3 we continue expanding the suspended nodes in the search tree with answer substitutions from the table. This is denoted by dotted lines in the search tree in Figure 4. Expansion of node 3 with the solution sub zero int from the table, introduces the new subgoal sub int T, which is added to the table together with the answers derived for it. Expansion of node 8 and node 5 does not yield any new answers. The table of stage 3 is depicted in Figure 5. In stage 4, no new answers can be derived by extending the search tree with the solutions from the previous stages. The search is saturated and all possible supertypes of the type zero have been inferred. Note that there are many more proofs to the query sub zero T that will not be generated. However a type-checker that executes the subtyping relation as a sub-routine should not distinguish between different proofs that the subtyping relation is satisfied. When executing the query sub zero T, we do not care about all possible proofs, but only about answers for T. For each answer, we are interested in a certificate (proof term) that can be checked, but all additional derivations that produce the same answers zero , nat and int are irrelevant. In contrast, if we ask what are the possible proofs for deriving sub zero zero then it is important to generate all possible derivations and search based on memoization is incomplete. However, usually we are interested in the existence of a proof and not in generating all possible proofs.

Entry	Answer	Ptr
sub zero T	[zero $/T$]	2
	[nat $/T$]	
	[int $/T$]	
sub nat T	[nat $/T$]	0
	[int $/T$]	

Entry	Answer	Ptr
sub zero T	[zero $/T$]	3
	[nat $/T$]	
	[int $/T$]	
sub nat T	[nat $/T$]	2
	[int $/T$]	
sub int T	[int $/T$]	0

Fig. 5. Table in stage 2 and stage 3

Although the idea of memoizing results of subcomputations is simple, it is challenging to realize it in practice. One crucial question is when to suspend nodes and retrieve answers. For example, if we reuse answer substitutions as soon as they are available and instead of suspending node 3, we expand it further using the answer [zero $/T$], then we might miss the answer [int $/T$]. The order in which we suspend nodes, awake suspended nodes and retrieve answers from the table critically influences the search, and not all computation strategies are complete. In the example above we awakened the suspended goals in the order they were suspended and we restricted the reuse of answers to solutions from previous stages. This strategy is called *multi-stage* strategy and is complete [47]. The success of memoization critically depends on efficiently accessing the table. Although we only derive and store relevant subgoals, the size of the table can grow to up to thousands of table entries and suspended goals. Therefore it is critical to design indexing data-structures, which allow us to efficiently store, lookup and retrieve table entries. This is also known as the *table access problem* [40].

2.4 Proof search

Proof search lies at the heart of logic programming interpreters and theorem provers that reason about specifications. In both cases we search for a proof given a set of clauses and axioms. However, our objectives might vary and different design decisions might be acceptable. In this section, we briefly discuss the proof search issues in logic programming and when reasoning with and about specifications.

Logic programming interpreter Computation in logic programming is achieved through proof search. To obtain a logic programming interpreter, a program should have a clearly defined procedural semantics. i.e. how is the program executed. This allows the programmer to predict and analyze the behaviour of the programs. Therefore we make several choices when designing a logic programming interpreter: First, we usually use depth-first search to derive a goal from a set of program clauses. Second, we try the program clauses in the order they were specified. Third, we process the subgoals of a clause in the order specified. As a consequence the search strategy of logic programming interpreters is incomplete and may not terminate although a proof exists. It has been often argued that not until we make these decisions is it possible to write an efficient and predictable logic programming interpreter. Programmers are usually aware of the procedural interpretation and exercise care to write executable programs. However, this might not always be trivial nor desirable under the traditional logic programming paradigm. The subtyping specification given in Figure 1 is straightforward, however execution with the traditional logic programming interpreter will not terminate and fails to enumerate all supertypes. Although it is possible to design a more efficient subtyping algorithm that executes correctly with traditional logic programming interpreter, this might not always be easy. Moreover, when executing the type-checker under the traditional logic programming interpretation, it will repeatedly evaluate subterms that occur multiple times in the expression. Using the tabled logic programming interpreter we often can obtain algorithms with better complexity.

Another example illustrating the benefits of tabled logic programming is the specification of context-free grammars [52]. Using a logic programming interpreter we then can check whether a given input string is accepted by the grammar or not. Many elegant context-free grammar specifications involve left-right recursion whose execution does not terminate under the traditional logic programming semantics. Even if our grammar is right-recursive and is executable with traditional logic programming, we obtain a recursive descent parser whose complexity is exponential. With tabled logic programming, one gets a recognition algorithm that is a variant of Early's algorithm (also known as active chart recognition algorithm) whose complexity is polynomial in the size of the input expression.

Tabled logic programming allows the programmer to implement simpler solutions and execute programs more efficiently. This additional power comes at a price: it might be more complicated to understand how the program will execute. This trade-off between additional functionality and more efficient execution on the one hand and a more complicated semantics on the other hand is also well-known in the context of programming languages. Lazy functional languages allow us to write programs where an argument is evaluated by need. Moreover, if the same expression occurs multiple times, then the expression is evaluated once, its value is memoized and its result is re-used. Hence, some programs with redundant computation run more efficiently. However, in contrast to strict functional programming, it is more complicated to understand the operational semantics of lazy functional programs.

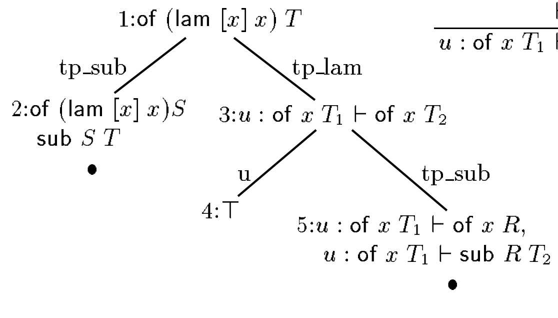
To take full advantage of the benefits it is important that tabled logic programs have a precise and high-level semantics that is compatible with traditional logic programming. Moreover, the programmer should be able to develop an intuitive understanding without much effort. This will play a central role when optimizing a tabled logic programming interpreter.

Theorem proving When reasoning with and about logical specifications, our objectives are different from logic programming. In particular, the restrictions imposed by the procedural semantics of the logic programming interpreter might be too severe. An unfair depth-first search strategy might be acceptable in the context of logic programming, but does not suffice when reasoning with specifications. Therefore, the theorem-prover *Twelf* employs bounded iterative deepening search to reason with specifications. As in logic programming, redundant computation hampers the performance of the search procedure. As we specify a depth bound for the iterative deepening search, the search does fail for branches that are potentially infinite in the search tree, however the user does not know whether the bound was too low or there exists no proof for the conjecture. In tabled search failure implies there exists no proof for the conjecture, at least for programs with bounded recursion. This enables the user to analyze failure and debug the proof attempt using the information in the table. To limit backtracking in the iterative deepening prover, the order in which clauses and subgoals are tried is fixed in the theorem prover. This has some intricate consequences. To illustrate we consider lemma application. Lemmas are of the form "if $\mathcal{D}$ and $\mathcal{E}$ then $\mathcal{F}$". When lemmas are used, the lemma specification is translated into higher-order logic programming clause $\mathcal{F} \leftarrow \mathcal{D} \leftarrow \mathcal{E}$. The order in which we specify the assumption $\mathcal{D}$ and $\mathcal{E}$ in the lemma not only influences the proof of the lemma itself, but also the order of subgoals later when the lemma is used. However, these two orderings might be incompatible and it seems unacceptable to require the user to understand and exploit these intricacies for finding a proof.

Therefore, a more robust search procedure that leads to more efficient performance of the theorem prover and better failure behaviour is clearly desirable.

2.5 Tabled higher-order logic programming

In this section, we highlight the challenges faced when designing a memoization-based semantics for higher-order logic programming. As briefly mentioned before, clauses in higher-order logic programming might contain nested implications and quantifiers. An example exhibiting both features is the clause tp_lam. Let us consider what happens when evaluating the query of (lam $[x]$ x) T (see Figure 6).


1:of (lam $[x]$ x) T
tp_sub
tp_lam
2:of (lam $[x]$ x)S
sub S T
•
3:u : of x T_1 ⊢ of x T_2
u
tp_sub
4:⊤
5:u : of x T_1 ⊢ of x R,
u : of x T_1 ⊢ sub R T_2
•

Entry	Answer	Ptr
⊢ of (lam $[x]$ x) T	$[P \rightarrow P/T$	0
u : of x T_1 ⊢ of x T_2	$[P/T_1, P/T_2]$	0

Fig. 6. Search tree after stage 1

The first observation is that not all answer substitutions are ground. For example, the first answer substitution for the query $\mathsf{of}\ (\mathsf{lam}\ [x]\ x)\ T$ is $[P \rightarrow P/T]$ where P is free. This phenonemon is not unique to higher-order logic programming, but already occurs in evaluating Horn clauses. The second observation is that goals occur in a dynamic context of assumptions. The possibility of nested implications and universal quantifiers however adds a new degree of complexity to memoization-based computation. This means we need to store goals together with the corresponding context in the table (see for example node 3). Proof terms and answer substitutions might also depend on the dynamic context. For example, the first answer for node 3 is obtained by using the assumption $u : \mathsf{of}\ x\ T_1$ to solve the goal $\mathsf{of}\ x\ T_2$. An entry now consists of the dynamic context Γ and the goal A. There are two basic choices how to lookup entries and retrieve answers for a given subgoal. 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 G against the goal G' 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.

The decision to store goals together with the context they occur in changes the nature of the table. The operations of storing, looking up and retrieving table entries and their answers have to be revised. There are essentially two critical optimizations. One optimization we already employed in the search tree above. As variables can stand for arbitrary higher-order terms, variables might depend on assumptions from the context Γ. For example, the new variables T_1 and T_2 in node 3 and R in node 5 might depend on the assumptions in Γ, i.e. they might depend on the new parameter x and the assumption u. However, a variable T can never depend on expressions or on the typing relation of . Using type dependency analysis based on subordination [50] 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.

To date, we have designed and implemented a prototype of a higher-order logic programming interpreter for *Elf* that memoizes sub-computations. Although we have not incorporated any indexing techniques for accessing the table, preliminary experiments indicate that many more specifications are executable. For example, we can execute the subtyping relations given in this proposal. In addition, evaluation is complete and terminates for programs that have the bounded-term size property. The table allows the programmer to analyze failure and debug the specifications. Computation for higher-order logic programs involves several challenges. From a practical perspective, it is important to find the right balance between efficiency of computation, overhead of storing, retrieving and looking up table entries and their answer substitutions. Different trade-offs might be appropriate for logic programming and theorem proving. From a theoretical perspective we propose a proof-theoretical characterization of tabled computation to provide a foundation for tabled higher-order logic programming.

3 Related work

3.1 Tabled logic programming

The problem of infinite and redundant paths of computation in the traditional logic programming interpretation has stimulated a large body of work in the first-order logic programming community. The goal of this work has been to design a tabled logic programming interpreter that executes with the speed of a Prolog interpreter and allows to mix Prolog and tabled logic programs. The most powerful interpreter, which attempts to remedy the infinite and redundant path problem, is the XSB system [43]. The XSB system is based on SLG resolution (linear resolution with selection function for general logic programs), which differs only insignificantly from the presented resolution strategy for programs without negation. It has been widely used in several interesting applications such as natural language processing, parsing [52], modeling concurrent processes and model checking [8]. Recently it has also been proposed as the foundation of model-carrying code for safety policies [46]. A large focus of the XSB work has been the design, implementation and evaluation of efficient indexing techniques [39, 12, 11, 10]. The tabling strategy implemented in the XSB system is based on SCC scheduling (strongly connected components), which allows us to consume answers as soon as they are available but not compromising completeness.

More recently, there has been interest in not only generating an answer for a given query, but also a justification for the answer [42, 17]. In this context, justifiers are directed acyclic graphs corresponding to the call graph where all leaves are either marked by "fail"(execution failed), "ancestor" (execution loops) or "fact" ($\top$). Justifiers are constructed by inspecting the program together with the computed memo tables. We can then efficiently reconstruct the proof (or sufficient evidence for the lack of a proof) for a goal for tabled logic programs. However, for parts that are not tabled, the program needs to be re-executed. Speculative justifiers evaluate the truth of literals in tandem with the justification. In higher-order tabled logic programming, proof terms play the role of justifiers. Currently, they are computed and stored together with the answer substitution for a given query.

3.2 Memoization in propositional theorem proving

In theorem proving for intuitionistic logic, memoization or loop detection has been investigated by Howe [20] among others. He designed a proof search procedure with loop detection for intuitionistic propositional calculus. While propositional logic allows nested implications, it does not deal with universal quantifiers. Howe designed a bottom-up proof procedure that carries a table to store intermediate formulas. If we need to prove a goal A and B then we use the table T to prove formula A and we use T to prove formula B. However there is no sharing between these two branches in the proof. This means if A and B share the sub-formula F, then we prove F (at least) twice. To handle changing context, Howe proposes to dispose the table, any time the context changes. The proposed work departs from this approach in two ways. First, the proposed search method handles universal quantification. Second, it is more ambitious in re-using results of subcomputation, as we allow maximum sharing across every connective.

3.3 Theorem proving in logical frameworks

A logical framework is a meta-language for the specification of deductive systems. A number of different frameworks have been proposed such as *Elf* based on LF type theory and λProlog [25, 14] or Isabelle [28, 29] based on hereditary Harrop formulas. For supporting theorem proving in these frameworks, two main approaches exist. The one presented so far uses search for a proof based on logic programming interpretation. *Elf* is one example of this category. The other approach pursued in λProlog and Isabelle 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. In this proposal, we plan to endow the logical framework with a tabled logic programming semantics. It forms the basis of an efficient general search strategy for specifications written in the logical framework.

3.4 Foundations for higher-order logic programming

Miller *et al.* [23] developed a proof-theoretic characterization of logic programming. Computation proceeds by goal-directed search that respects the interpretation of logical connectives as search instructions. It not only serves as a foundation for Prolog-like first-order logic programming, but also for higher-order logic programming languages such as *Elf* or *λProlog*. To search for a proof of a goal, interpreters *Elf* or *λProlog* uses depth-first search. The theorem prover *Twelf* is based on iterative deepening [44]. Based on uniform proofs, Cervesato [4] developed a proof-theoretic view of compilation for logic programming languages. However so far there has been no effort to develop a proof-theoretic foundation for tabled logic programming. Tamaki and Sato designed tabled resolution as a refinement of the traditional SLD resolution. The formal presentations of tabled resolution are based on a highly procedural semantics of Horn clauses. This forms the foundation of soundness and completeness proofs of tabled resolution for Horn clauses.

The development of SLG resolution used in the XSB system is also highly procedural. Descriptions of the corresponding abstract machine are based on an extensions of the

WAM (Warren abstract machine), the SLG-WAM, and often are presented as low-level instructions operating on freeze and choice point registers.

4 Overview of proposed work

I propose to develop a tabled higher-order logic programming interpretation for the logical framework LF. This will allow us to execute more specifications and run implementations more efficiently. As tabled higher-order logic programming semantics is also the foundation for reasoning *about* logical specifications, it potentially leads to more efficient meta-theorem proving. If there are finitely many answers, the search will terminate. In addition, the search tree can be inspected with the table at hand, which is useful for debugging specifications and proofs.

In contrast to first-order logic programming, which is based on Horn clauses, higher-order logic programming based on hereditary Harrop formulas or based on LF allows nested implications and universal quantifiers. A major contribution of the work to date is the design and implementation of a higher-logic programming interpreter based on memoization. The key idea is to store goals together with a context of assumptions in the table. Preliminary experiments demonstrate that many more logical specifications are executable and implementations of type-checkers run more efficiently.

Efficiently evaluating a higher-order logic program given a set of additional assumptions, also plays a critical role when reasoning *about* logical specifications. Therefore, an important issue is to incorporate memoization-based computation in the meta-theorem prover *Twelf* [45,44]. *Twelf* essentially loops over three main tasks: search based on iterative deepening, case analysis and induction hypothesis generation. One bottleneck of the current prover is the search procedure. It seems possible to use search based on memoization as a center piece in the meta-theorem prover. The table survives case analysis and induction hypothesis and is reused and extended in each iteration of the loop. This has the potential of improving the performance of the meta-theorem prover. In addition, tabled execution promises better failure behaviour. If the size of all subgoals for any derivation can be bound, then we say the program fulfills the bounded term size property or the program recursion is bounded. In this case if search fails, we know that there exists no proof. If there are infinitely many answers possible, heuristics can ensure that answers are generated fairly.

As mentioned before, the success of memoization critically depends on efficiently accessing the table, i.e. looking up table entries, inserting new answers and retrieving answers. A wide variety of indexing data structures (for a survey see [39]) have been developed to support fast retrieval of terms, i.e. for any given query term t, retrieve all terms in the index satisfying a certain relation with t, such as matching, unifiability, or syntactic equality. To minimize the retrieval time, the aspect of memory consumption seems to become more and more important [15]. Excessive memory consumption leads to more cache misses, which become the dominant factor for retrieval time. In the context of *Twelf*, indexing techniques will not only lead to a more compact table representation, but also can be used to index the logic program itself.

To keep the table size small, backward and forward subsumption is desirable. Substitution trees [16] are ideally suited as they efficiently support both operations. Klein [21] describes indexing of higher-order terms for the simply typed λ-calculus using substitution trees in his master thesis.

Another critical issue is the efficient handling of proof terms, as they can be quite large and increase the size of the table dramatically. Indexing techniques can lead to a more compact representation of proof terms, however it might be more efficient to store only proof term skeletons and the reconstruct the proof term with the table at hand. Such techniques have been investigated in the context of proof and model carrying code [26, 42].

In this proposal, we concentrate on the proof-theoretic characterization of tabled logic programming. We describe the relationship between tabled and non-tabled logic programming based on uniform proofs. It is also the basis of the implemented prototype for tabled logic programming interpreter for *Elf*.

5 A foundation for tabled higher-order logic programming

In this section we formally describe the search semantics based on memoization. As a starting point, we will review briefly uniform deductions $\mathcal{L}$ and then present the search semantics $\mathcal{L}_s$ with answer substitutions. Finally, we will extend $\mathcal{L}_s$ to search semantics $\mathcal{LT}_s$ that incorporates memoization.

5.1 Uniform proofs

Computation in logic programming is achieved through proof search. Given a goal (or query) A and a program Γ, we want to derive A by successive application of clauses of the program Γ. Miller *et al* [23] propose to interpret the connectives in a goal A as *search instructions* and the clauses in Γ as specifications of how to continue the search when the goal is atomic.

A proof is *goal-oriented* if every compound goal is immediately decomposed and the program is accessed only after the goal has been reduced to an atomic formula. A proof is *focused* if every time a program formula is considered, it is processed up to the atoms it defines without need to access any other program formula. A proof having both these properties is *uniform* and a formalism such that every provable goal has a uniform proof is called an *abstract logic programming language.*

The largest freely generated fragment of intuitionistic logic that constitutes an abstract programming language, contains conjunctions, universal quantifiers and implications. This choice is not accidental, but motivated by the fact that all the introduction rules for these connectives are invertible. Therefore, goal formulas that only consist of these connectives can be eagerly processed up to the atoms. *Elf* is one example of an abstract logic programming language, which is based on the LF type theory. In this setting types are interpreted as clauses and goals and typing context represents the store of program clauses available. We will use types and formulas interchangeably. Types and programs are defined as follows by means of the following grammar:

$$\begin{array}{ll} \text{Types } A & ::= a \mid \top \mid A_1 \wedge A_2 \mid A_1 \to A_2 \mid \Pi x : A_1.A_2 \\ \text{Programs } \Gamma & ::= \cdot \mid \Gamma, x : A \end{array}$$

a ranges over atomic formulas. The function type $A_1 \to A_2$ corresponds to an implication. The Π-quantifier, denoting dependent function type, can be interpreted as the universal $\forall$-quantifier. To describe only the fragment corresponding to LF type theory, Π-quantifier and $\to$ suffice. We extend this fragment with (additive) product type $\wedge$, which corresponds to a conjunction, and the (additive) unit type $\top$ corresponding to true. The clause `tr : sub T S <- sub T R <- sub R S.` is interpreted as $\mathsf{tr}{:}\Pi t{:}\mathsf{tp}.\Pi s{:}\mathsf{tp}.\Pi r{:}\mathsf{tp}.\ \mathsf{sub}\ r\ s \to (\mathsf{sub}\ t\ r \to \mathsf{sub}\ t\ s)$. Operationally, it will behave equivalently to $\mathsf{tra}{:}\Pi t{:}\mathsf{tp}.\Pi s{:}\mathsf{tp}.\ \Pi r{:}\mathsf{tp}.\mathsf{sub}\ t\ r \wedge \mathsf{sub}\ r\ s \to \mathsf{sub}\ t\ s$.

Extending the core fragment corresponding to LF with $\wedge$ and $\top$ has mainly two reasons. From a logic programming point of view, it might be more intuitive to think of the subgoals in a clause $H \leftarrow A_1 \leftarrow \ldots \leftarrow A_n$ as a conjunction $A_1 \wedge \ldots \wedge A_n$. From a type-theoretic view, these additional constructors enrich the LF type theory. To obtain a linear type theory including additive products, additive unit and linear and non-linear function types [5], we add the linear function type. We hope that this underscores the applicability of this approach to other logic programming languages such as linear logic programming.

Every type has a corresponding proof term M. To represent proof terms, we will use the spine notation [6]. We assume all proof terms are in normal form.

$$\begin{array}{lll}
\text{Terms } M & ::= & H \cdot S \mid \lambda x : A.M \mid \langle M_1, M_2 \rangle \mid \langle\rangle \\
\text{Spines } S & ::= & \textsc{nil} \mid M; S \mid \pi_1\ S \mid \pi_2\ S \\
\text{Heads } H & ::= & c \mid x
\end{array}$$

To illustrate this notation we give a few examples of proof terms in conventional notation, types and the proof term in spine notation. In the example from Section 5.3, the proof term corresponding to $\mathsf{sub\ zero\ int}$ is given as $\mathsf{tr\ zn\ nati}$. Note that we actually omitted the implicit arguments zero , nat and int , which denote the instantiation of transitivity rule. In the following examples, we will include implicit arguments in the proof term representation.

Proof term	Type	Spine notation
nati	$\mathsf{sub\ nat\ int}$	$\mathsf{nati} \cdot \textsc{nil}$
$\mathsf{tr\ zero\ \ int\ \ nat\ \ zn\ nati}$	$\mathsf{sub\ zero\ int}$	$\mathsf{tr}\ \cdot \mathsf{zero}\ ;\ \mathsf{int}\ ;\ \mathsf{nat}\ ;\ \mathsf{zn}\ ; \mathsf{nati}\ ; \textsc{nil}$
$\mathsf{tra\ zero\ \ int\ \ nat}\ \langle \mathsf{zn}, \mathsf{nati} \rangle$	$\mathsf{sub\ zero\ int}$	$\mathsf{tra}\ \cdot \mathsf{zero}\ ;\ \mathsf{int}\ ;\ \mathsf{nat}\ ;\ \langle \mathsf{zn} \cdot \textsc{nil}, \mathsf{nati} \cdot \textsc{nil} \rangle\ ; \textsc{nil}$
$\mathsf{tp\_lam}\ T\ (\lambda x : \exp .x)\ T$ $(\lambda x : \exp .\lambda u : \mathsf{of}\ x\ T.\ u)$	$\mathsf{of}\ (\mathsf{lam}\ \lambda x : \exp .x)(T \to T)$	$\mathsf{tp\_lam} \cdot T\ ;\ (\lambda x : \exp .x \cdot \textsc{nil})\ ;\ T\ ;$ $(\lambda x : \exp .\lambda u : \mathsf{of}\ x\ T.\ u \cdot \textsc{nil})\ ; \textsc{nil}$

We can characterize uniform proofs by two main judgments.

$$\begin{array}{lrl}
\text{Judgments:} & \Gamma \xrightarrow{u} M : A & \text{uniform proof} \\
 & \Gamma \gg A \xrightarrow{f} S : a & \text{focused proof}
\end{array}$$

Γ represents the program while A represents the goal, which we need to derive from clauses in Γ. M denotes the the proof term corresponding to A. We will use the first judgment to describe decomposition of the goal A. The judgment $\Gamma \xrightarrow{u} M : A$ says there is a uniform proof for A with proof term M from the program context Γ. Taking a type-theoretic view, it means, M has type A in the context Γ. Once A is atomic, we will focus on a clause from Γ and transition to the second judgment. The judgment $\Gamma \gg A \xrightarrow{f} S : a$ says there exists a focused proof for the atom a with spine S and A is a clause from Γ on which we focus. The inference rules for $\xrightarrow{f}$ will break down the clause A until A is atomic. Once the focused formula A on the left-hand side and the goal on the right-hand side are atomic and they coincide, we can close the branch. Inference rules describing uniform proofs are given in Figure 7.

$$\frac{\Gamma, x : A, \Gamma' \gg A \xrightarrow{f} S : a}{\Gamma, x : A, \Gamma' \xrightarrow{u} x \cdot S : a}\,\textsf{u\_atom} \qquad \frac{}{\Gamma \gg a \xrightarrow{f} \textsc{nil} : a}\,\textsf{f\_atom}$$

$$\frac{\Gamma, c : A_1 \xrightarrow{u} [c/x]M : [c/x]A_2}{\Gamma \xrightarrow{u} \lambda x : A_1.M : \Pi x : A_1.A_2}\,\textsf{u\_forall}^c \qquad \frac{\Gamma \gg [M/x]A_2 \xrightarrow{f} S : a \qquad \Gamma \xrightarrow{u} M : A_1}{\Gamma \gg \Pi x : A_1.A_2 \xrightarrow{f} M; S : a}\,\textsf{f\_forall}$$

$$\frac{\Gamma, x : A_1 \xrightarrow{u} M : A_2}{\Gamma \xrightarrow{u} \lambda x : A_1.M : A_1 \to A_2}\,\textsf{u\_imp}^u \qquad \frac{\Gamma \gg A_1 \xrightarrow{f} S : a \qquad \Gamma \xrightarrow{u} M : A_2}{\Gamma \gg A_2 \to A_1 \xrightarrow{f} M; S : a}\,\textsf{f\_imp}$$

$$\frac{\Gamma \xrightarrow{u} M : A_1 \qquad \Gamma \xrightarrow{u} N : A_2}{\Gamma \xrightarrow{u} \langle M, N \rangle : A_1 \wedge A_2}\,\textsf{u\_and} \qquad \frac{\Gamma \gg A_1 \xrightarrow{f} S : a}{\Gamma \gg A_1 \wedge A_2 \xrightarrow{f} \pi_1 S : a}\,\textsf{f\_and}_1$$

$$\frac{}{\Gamma \xrightarrow{u} \langle\rangle : \top}\,\textsf{u\_true} \qquad \frac{\Gamma \gg A_2 \xrightarrow{f} S : a}{\Gamma \gg A_1 \wedge A_2 \xrightarrow{u} \pi_2 S : a}\,\textsf{f\_and}_2$$

Fig. 7. Uniform deduction system for $\mathcal{L}$

In the rule f_forall, we instantiate the bound variable x with a term M. As x has type A_1, we check that M has type A_1 in Γ. Miller [22] shows for the simply-typed λ calculus that if M is a solution for x in the context Γ then there exists a solution M' of type $\Pi\Gamma.A_1$ such that $M' \cdot \Gamma$ is also a solution for x and M' is well-typed in the empty context. We write $\Pi\Gamma.A_1$ for the type $\Pi x_1 : B_1, \ldots, x_n : B_n.A_1$ where Γ is a context $x_1 : B_1, \ldots, x_n : B_n$ and $M' \cdot \Gamma$ as an abbreviation for $M' \cdot x_1; \ldots; x_n$. Moreover, there is a one-to-one correspondence between these two solutions. Following Miller's terminology, we say M' is the result of raising M. Pfenning [31] investigated this notion in the setting of the calculus of construction, which includes LF. We will rewrite the f_forall rule to reflect this view to:

$$\frac{\Gamma \gg [M' \cdot \Gamma/x]A_2 \xrightarrow{f} S : a \qquad \cdot \xrightarrow{u} M' : \Pi\Gamma.A_1}{\Gamma \gg \Pi x : A_1.A_2 \xrightarrow{f} (M' \cdot \Gamma); S : a}\ \textsf{f\_forall}$$

The proof term represents the witness of the proof. When searching for a uniform proof, the proof term is constructed simultaneously as a certificate of the actual proof. In the following discussion, we will not mention proof terms explicitly, but keep in mind that they are silently generated as a result of the proof.

5.2 Uniform proofs with answer substitutions

The result of a computation in logic programming is generally something that is extracted from the proof of a goal A given a program Γ. Typically this extraction is a substitution θ, called an *answer substitution*, for the existentially quantified variables in the goal. To obtain an algorithm that computes answer substitutions, we substitute existential variables X for the bound variable x in the f_forall rule. Existential variables are instantiated later during unification yielding a substitution θ. To model dependencies between parameters and existential variables, we will annotate existential variables X with their type. An alternative would be to use mixed-prefixes [22] to model dependencies. However this would complicate the presentation further. We view the answer substitution θ as a collection of constraints to the existential variables in a goal A. In general, unification for higher-order terms is undecidable, however Pfenning showed that unification of higher-order patterns in the context of LF type theory is decidable and unitary[31]. The constraints in θ essentially describe the solution to a higher-order pattern unification problem. However, note that we can extend the notion of constraints to the general case.

Substitution $\theta ::= \cdot | \theta, X_A = M$

We require that all free (existential) variables X defined by a substitution are distinct. We write $\mathsf{dom}(\theta)$ for the free variables defined by a substitution and $\mathsf{codom}(\theta)$ for all the free variables occurring in the term M. For a *ground substitution* $\mathsf{codom}(\theta)$ is empty. We will write $M[\theta]$, $A[\theta]$, and $\Gamma[\theta]$ for the application of a substitution to a term, proposition or context. *Composition*, written as $\theta_1 \circ \theta_2$, has the property that $M[\theta_1 \circ \theta_2] = (M[\theta_1])[\theta_2]$ and similarly for propositions and contexts. Composition is defined as follows:

$$\begin{gathered} \cdot \circ \theta = \theta \\ (\theta_1, X_A = M) \circ \theta_2 = (\theta_1 \circ \theta_2), X_{A[\theta_2]} = M[\theta_2] \end{gathered}$$

In order for composition of substitutions to be well-defined and have the desired properties we require that $\mathsf{dom}(\theta_1)$ and $\mathsf{dom}(\theta_2)$ are disjoint, but of course variables in the co-domain of θ_1 can be defined by θ_2. Moreover, we require that $\theta \circ \theta = \theta$, which implies that $\mathsf{dom}(\theta)$ and $\mathsf{codom}(\theta)$ are disjoint. As an existential variable is annotated with its type A and A might itself contain existential variables, we need to apply the substitution θ_2 to M and to the type A during composition of substitutions.

The two main judgments for computing answer substitutions are as follows:

$$\text{Judgments:} \qquad \begin{array}{r} \Gamma \xrightarrow{u} A/\theta \\ \Gamma \gg A \xrightarrow{f} a/\theta \end{array}$$

The inference rules are given in Figure 8. To obtain an algorithm, we impose left-to-right order on the solution of the us_and rule and fs_imp rule. This matches our intuitive understanding of computation in logic programming. In the fs_imp rule for example we first decompose the focused clause until we reach the head of the clause. After we unified the head of the clause with our goal A on the right-hand side of the sequent and completed this branch, we proceed proving the subgoals. This left-to-right evaluation strategy only fixes a don't care non-deterministic choice in the inference system. In the fs_forall rule we delay the instantiation of x by introducing an new existential variable X. In the fs_atom rule the instantiation for existentially quantified variables is obtained by unifying a with a' in the context Γ. θ is a solution to the unification problem $\Gamma \vdash a \doteq a'$.

$$\frac{\Gamma, x : A, \Gamma' \gg A \xrightarrow{f} a/\theta}{\Gamma, x : A, \Gamma' \xrightarrow{u} a/\theta}\ \mathsf{us\_atom} \qquad\qquad \frac{\Gamma \vdash a' \doteq a/\theta}{\Gamma \gg a' \xrightarrow{f} a/\theta}\ \mathsf{fs\_atom}$$

$$\frac{\Gamma, c : A_1 \xrightarrow{u} [c/x]A_2/\theta}{\Gamma \xrightarrow{u} \Pi x : A_1.A_2/\theta}\ \mathsf{us\_forall}^c \qquad \frac{\Gamma \gg [X_{\Pi\Gamma.A_1} \cdot \Gamma/x]A \xrightarrow{f} a/\theta \qquad X_{\Pi\Gamma.A_1} \text{ is new}}{\Gamma \gg \Pi x : A_1.A_2 \xrightarrow{f} a/\theta}\ \mathsf{fs\_forall}$$

$$\frac{\Gamma, u : A_1 \xrightarrow{u} A_2/\theta}{\Gamma \xrightarrow{u} A_1 \to A_2/\theta}\ \mathsf{us\_imp}^u \qquad \frac{\Gamma \gg A_1 \xrightarrow{f} a/\theta_1 \qquad \Gamma[\theta_1] \xrightarrow{u} A_2[\theta_1]/\theta_2}{\Gamma \gg A_2 \to A_1 \xrightarrow{f} a/\theta_1 \circ \theta_2}\ \mathsf{fs\_imp}$$

$$\frac{\Gamma \xrightarrow{u} A_1/\theta_1 \qquad \Gamma[\theta_1] \xrightarrow{u} A_2[\theta_1]/\theta_2}{\Gamma \xrightarrow{u} A_1 \wedge A_2/\theta_1 \circ \theta_2}\ \mathsf{us\_and} \qquad \frac{\Gamma \gg A_1 \xrightarrow{f} a/\theta}{\Gamma \gg A_1 \wedge A_2 \xrightarrow{f} a/\theta}\ \mathsf{fs\_and}_1$$

$$\frac{}{\Gamma \xrightarrow{u} \top/\cdot}\ \mathsf{us\_true} \qquad\qquad \frac{\Gamma \gg A_2 \xrightarrow{u} a/\theta}{\Gamma \gg A_1 \wedge A_2 \xrightarrow{f} a/\theta}\ \mathsf{fs\_and}_2$$

Fig. 8. Uniform deduction system for $\mathcal{L}_\theta$ with substitutions

5.3 Tabled uniform proofs

In this section, we will extend the deductive system $\mathcal{L}_s$ to allow memoization. We will keep in mind that we are silently generating proof terms as part of the answer substitution. The idea is to extend our two basic judgments with a table $\mathcal{T}$ in which we record atomic sub-goals and the corresponding answer substitutions and proof terms. A subgoal is a sequent $\Gamma \xrightarrow{u} a$ where Γ is a program context and a is an atomic goal, which we need to derive from Γ. When we discover the sub-goal $\Gamma \xrightarrow{u} a$ for the first time, we memoize this goal in the table. Note that the sequent $\Gamma \xrightarrow{u} a$ might potentially contain existential variables. Once we have proven the sub-goal $\Gamma \xrightarrow{u} a$, we add the answer substitution θ to the table.

Definition 1 (Table). *A table $\mathcal{T}$ is a collection of table entries. A table entry consists of two parts: a goal $\Gamma \xrightarrow{u} a$ and a list $\mathcal{A}$ of answer substitutions θ such that $\Gamma[\theta] \xrightarrow{u} a[\theta]$ is a solution.*

We will design the inference rules in such a way that for any solution in the table $\Gamma[\theta] \xrightarrow{u} a[\theta]$ there exists a derivation $\Gamma \xrightarrow{u} a/\theta$. We will keep all the previous inference rules, but keep in mind that we are silently passing around a table $\mathcal{T}$. This also means that any substitution we apply to Γ and a (see for example the us_and or the fs_imp rule) will not effect the table. This is important because we do want to have explicit control over the table. The application of inference rules should not have any undesired effects on the table. In addition to these inference rules, we will add two more rules for storing a subgoal and its answer substitution in the table and for retrieving an answer substitution for a subgoal. The main judgments are as follows:

$$\begin{array}{ll} \text{Judgments} & \mathcal{T}; \Gamma \xrightarrow{u} A/(\theta, \mathcal{T}') \\ & \mathcal{T}; \Gamma \gg A \xrightarrow{f} a/(\theta, \mathcal{T}') \end{array}$$

In addition to the us_atom inference rule, we will have the rules extend and retrieve. extend adds a subgoal and its answer to the table. retrieve allows us to close a branch by unifying the current goal with the solution from the table. We assume that some predicates are designated as tabled predicates where we apply the rules extend and retrieve to record subgoals and corresponding answers. Predicates not designated as tabled predicates are treated as usual using rule us_atom .

We consider $\Gamma \xrightarrow{u} a$ a variant of $\Gamma' \xrightarrow{u} a'$ if there exists a renaming of the bound and existential variables such that $\Gamma \xrightarrow{u} a$ is equal to a'.

Definition 2 (Variant).
The goal $\Gamma \xrightarrow{u} a$ is a variant of $\Gamma' \xrightarrow{u} a'$ if

- *there exists a bijection between the existential variables in $\Gamma \xrightarrow{u} a$ and $\Gamma' \xrightarrow{u} a'$*
- *there exists a bijection between the bound variables in $\Gamma \xrightarrow{u} a$ and $\Gamma' \xrightarrow{u} a'$.*

such that such that $\Gamma \xrightarrow{u} a$ is α-convertible to $\Gamma' \xrightarrow{u} a'$.

Now we can define the three main operations on the table, extending the table, inserting an answer in the table and retrieving an answer from the table.

$$\frac{\begin{array}{l}\mathsf{extend}(\mathcal{T}, (\Gamma, x : A, \Gamma') \xrightarrow{u} a) = \mathcal{T}_1 \\ \mathcal{T}_1; (\Gamma, x : A, \Gamma') \gg A \xrightarrow{f} a/(\theta, \mathcal{T}_2) \\ \mathsf{insert}(\mathcal{T}_2, (\Gamma, x : A, \Gamma') \xrightarrow{u} a, \theta) = \mathcal{T}_3\end{array}}{\mathcal{T}; (\Gamma, x : A, \Gamma') \xrightarrow{u} a/(\theta, \mathcal{T}_3)}\ \mathsf{extend}$$

$$\frac{\mathsf{retrieve}(\mathcal{T}; \Gamma \xrightarrow{u} a) = \theta}{\mathcal{T}; \Gamma \xrightarrow{u} a/(\theta, \mathcal{T})}\ \mathsf{retrieve}$$

Fig. 9. Memoization extensions

Definition 3 (extend). *extend*$(\mathcal{T}, \Gamma \xrightarrow{u} a) = \mathcal{T}'$

Let $\mathcal{T}$ be a table, $\Gamma \xrightarrow{u} a$ be a goal.

If there exists a table entry $(\Gamma' \xrightarrow{u} a', \mathcal{A})$ in $\mathcal{T}$ such that $\Gamma' \xrightarrow{u} a'$ is a variant of $\Gamma \xrightarrow{u} a$, and $\mathcal{A}$ is not empty then return $\mathcal{T}$.

If there exists **no** *table entry $(\Gamma' \xrightarrow{u} a', \mathcal{A})$ in $\mathcal{T}$ such that $\Gamma' \xrightarrow{u} a'$ is a variant of $\Gamma \xrightarrow{u} a$, then we obtain the extended table $\mathcal{T}'$ by renaming all the existential variables in $\Gamma \xrightarrow{u} a$ and adding the renamed goal to the table $\mathcal{T}$ with an empty solution list.*

By rename all existential variables before adding a goal to the table, we enforce a clear separation between the table and the goals discovered during the application of inference rules.

Definition 4 (insert). *insert*$(\mathcal{T}, \Gamma \xrightarrow{u} a, \theta) = \mathcal{T}'$
Let $\mathcal{T}$ be a table, $\Gamma \xrightarrow{u} a$ be a goal and θ be a corresponding answer substitution. Let $(\Gamma_i \xrightarrow{u} a_i, \mathcal{A})$ be in the table $\mathcal{T}$ and $\Gamma_i \xrightarrow{u} a_i$ is a variant of $\Gamma \xrightarrow{u} a$. If there exists θ_i in the answer substitution list $\mathcal{A}$, such that $\Gamma_i[\theta_i] \xrightarrow{u} a_i[\theta_i]$ is a variant of $\Gamma[\theta] \xrightarrow{u} a[\theta]$, then we fail otherwise we match $\Gamma_i \xrightarrow{u} a_i$ against $\Gamma[\theta] \xrightarrow{u} a[\theta]$ and add the resulting substitution θ' to $\mathcal{A}$.

Note that the result of matching $\Gamma_i \xrightarrow{u} a_i$ against $\Gamma[\theta] \xrightarrow{u} a[\theta]$ is a substitution θ' such that $\Gamma_i[\theta'] \xrightarrow{u} a_i[\theta']$ is a variant of $\Gamma[\theta] \xrightarrow{u} a[\theta]$. If we discover a sub-goal a in a context Γ that is already in the table $\mathcal{T}$ but with an empty answer substitution list, then we have discovered a loop in the computation. No inference rule is applicable, and therefore computation just fails. The definitions of extend and insert also prevent us from inferring the same solution twice.

If a sub-goal a is already in the table, but has some answers in the answer list $\mathcal{A}$, then we retrieve the answers. As we might need additional answers for a that are not already in the table yet, we need to still be able to apply extend rule. If we infer an answer for a that is already in the table, then we fail. In a real implementation we need to control the choice between extending the table and retrieving answers. One solution to this problem is to explore all search paths to prove a goal a by applying extension rule and backtracking. The proof-tree for a will only contain success and failure leaves. Then in the next step, we retrieve answers using the retrieve rule.

Definition 5 (retrieve). $\mathsf{retrieve}(\mathcal{T}, \Gamma \xrightarrow{u} a) = \theta$

Let $\mathcal{T}$ be a table and $\Gamma \xrightarrow{u} a$ be a goal. If there exists a table entry $(\Gamma_i \xrightarrow{u} a_i, \mathcal{A}_i)$ such that $\Gamma_i \xrightarrow{u} a_i$ is variant of $\Gamma \xrightarrow{u} a$, then match $\Gamma \xrightarrow{u} a$ against $\Gamma_i[\theta_i] \xrightarrow{u} a_i[\theta_i]$ to obtain a substitution θ.

The presented inference rules leave several choices undetermined, for example when to apply the retrieve rule and when to apply the extend rule. There is also non-determinism in the retrieve and us_atom rule. In the retrieve rule, we need to decide what answer substitution to select, if there is more than one. Similarly, in the us_atom rule, the order in which clauses are tried is left undetermined. In an actual implementation all these choices need to be resolved and there are several possible strategies. For example to resolve the non-determinism in the us_atom rule logic programming interpreters try the program clauses in the order they are specified. The multi-stage strategy described in Section 5.3 fixes the non-determinism on the extend and retrieve rule. If a variant of a previous subgoal with no answers is encountered, then search just fails in the presented inference rules. A different combination of clause application however might lead to success. In a real implementation we store suspended goals a together with their context Γ and pending sub-goals before we fail and avoid repeating this work. After some answers have been generated for the sequent $\Gamma \xrightarrow{u} a$, we awaken the suspended goal and resume computation of the pending sub-goals.

This proof-theoretic view on computation based on memoization offers several advantages. First, it provides a high-level description of a tabled logic programming interpreter. In fact, it is very close to our prototype implementation for *Elf*. It seems also plausible to adopt the techniques described here to other logic programming languages and theorem proving in general. To obtain a formulation for the linear dependently typed lambda calculus for example we add a linear function arrow. It seems also straightforward to adopt this search semantics to λProlog, which is based on hereditary Harrop formulas.

Second, we observe that some optimization such as substitution factoring [38] come up naturally in the proof-theoretic view, as we only store the answer substitutions θ. Ramakrishnan *et al.* [38, 40] propose substitution factoring to minimize the access cost for an answers. Using this technique, the access cost for answers is proportional to the size of the answer substitution, rather than to the size of the answer itself. This idea is supported in this theoretical framework. To implement substitution factoring efficiently, Ramakrishnan *et al* first standardize terms by numbering distinct variables and then add the standardized term to the table. Answer substitutions, interpreted as an ordered list of instantiations, are stored separately. In the current implementation of the prototype, we incorporated this technique.

We conjecture that search based on tabled uniform proofs is sound. Soundness of tabled uniform proofs implies the soundness of the actual implementation, independently of what strategies we choose to resolve the remaining non-determinism. With tabled uniform proof search we will find fewer proofs than in the uniform proof system $\mathcal{L}$. For example in the subtyping example given in Section the query sub zero T will have infinitely many proofs under the traditional logic programming interpretation. 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 [33]

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. In this setting where two proofs are considered equal if they produce the same answer, it seems plausible to show that search based on tabled uniform proofs is also non-deterministically complete, i.e. if computation fails, then there exists no proof.

6 Optimizations for efficient tabled computation

To achieve good performance of proof search based on memoization, it is critical to access the table efficiently. In this section we discuss several optimizations some of which we already implemented and some we propose to incorporate.

6.1 Scheduling strategies

Search based on memoization is critically influenced by when we retrieve answers and when we suspend goals and when and in what order we awaken suspended goals. Currently, we have implemented *multi-stage depth-first strategy*. The search strategy consists of multiple stages. In the i-th stage we are allowed to reuse answers from all previous stages 1 to $i-1$. If the table does not change anymore, the search is saturated. The advantage is that it is simple to control and to understand. However, we might delay finding an answer, because of the answer retrieval restriction. The XSB system therefore uses a strategy based on strongly connected components, which allows us to consume answers as soon as they are available. Using this strategy, we compute a subgoal dependency graph. This graph might fall into different connected components, separating subgoals that can and cannot influence each other. If computation for all the subgoals within a connected component is saturated, then we can dispose all the suspended sub-goal belonging to this component. As we are able to delete suspended nodes that cannot contribute to new solutions anymore, this leads to fewer suspended nodes and has the potential to be more space and time efficient. This might be advantageous especially in theorem proving, where we only care about one answer to the query and are not interested in mimicking Prolog execution.

The order in which we awaken suspended goals also critically influences the search space. Currently we awaken them in the order they were suspended. However, some other ordering on suspended goals might be more efficient.

6.2 Variant and Subsumption Checking

A basic question is how to check whether a goal $\Gamma \xrightarrow{u} a$ is already in the table. So far we have discussed variant-checking, where the goal in the table $\Gamma' \xrightarrow{u} a'$ is α-variant to the goal $\Gamma \xrightarrow{u} a$. In subsumption-based checking, we check whether the goal $\Gamma \xrightarrow{u} a$ is an instance of a sequent $\Gamma' \xrightarrow{u} a'$ in the table using matching. We have implemented variant and subsumption checking. Variant-based tabling preserves the behavior of Prolog, while subsumption-based method may have better termination and complexity properties for certain programs and queries. Research on first-order logic programming has advocated variant based checking as it is simpler and more efficient. In addition, it fits very well with traditional Prolog execution. Subsumption checking may lead to more compact and smaller tables, especially with backward and forward subsumption, but looking up whether an entry is already in the table requires higher-order matching, which is an expensive operation. Therefore, it is critical to understand the trade-off between variant and subsumption based checking.

6.3 Term and context depth suspension

Term depth abstraction has been proposed by Tamaki and Sato [47] to bound the number of distinct subgoals to be solved. Using term depth abstraction, every subterm of depth k is replaced by distinct new variables. The goal is to cut off branches in the search tree, which contain diverging sub-goals. If we use subsumption-based memoization, then in the next iteration the table will contain a more general goal than the current diverging goal, and the computation would be suspended. Instead of trying to solve the diverging goal, they propose to solve the abstracted goal, which is more general. Term depth abstraction acts as a safety valve and forces the search procedure to delay solving branches that contain diverging goals. It also makes computation more robust to subgoal reorderings. However, it may generate subgoals that are not directly relevant for solving the original query. For a start, we have implemented a mechanism that forces the suspension of a goal, if the term depth k is exceeded. This is a simpler mechanism than term depth abstraction, but can be used with either variant or subsumption based checking.

In addition, higher-order logic programming could benefit from context depth and context length suspension. Context depth is determined by the greatest term depth of one of the elements. This causes a branch to be suspended, if one of the elements in the context diverges. Context length is determined by the number of elements in the context. This is a precaution against extending the context with new elements without making any progress on the goal.

Term and context depth suspension are heuristics that control the search space. This is especially important if the program have infinitely many answers and sub-goals are diverging. It seems also possible to automatically discover diverging branches, and then apply term and context depth suspension to them. In induction theorem proving Walsh and Basin [51, 3] developed techniques for detecting divergence, which could be useful heuristics to control the search space.

6.4 Strengthening of hypothesis and contraction

Although subordination analysis is powerful, the context Γ might still contain assumptions that potentially can contribute to proving a goal G, but in fact are not used. To design even more aggressive table access operations, it might be desirable to incorporate strengthening and weakening on assumptions.

6.5 Interaction between tabled and non-tabled predicates

While tabled computation 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 tabling 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. Although the current prototype does not support the mix of tabled and non-tabled predicates, extensions required seem straightforward. The programmer will be required to designate some predicates as tabled.

7 Further Work

7.1 High-level optimizations

Although memoization-based computation aims to make reasoning *within* logical specification efficient, it cannot rival the performance of a theorem prover that is specifically built for a given logic. One reason is that specialized state-of-the-art theorem provers exploit properties of the theory they are built for. For example, inverse method theorem provers for first-order logic like Gandalf [48] exploit the subformula property. Other theorem provers like Spass [53], which are very successful in equational reasoning, rely on orderings to restrict the search. These meta-level optimizations can improve performance of higher-order logic programming dramatically. Therefore, one interesting path to explore is to verify such properties about logical specifications in advance and exploit them during search.

7.2 Properties about tabled logic programs

There has been a large body of work to verify properties of non-tabled logic programs such as well-modedness or termination. However, for tabled logic programs and mixed tabled/non-tabled logic programs only few automated techniques exist [13, 49]. Any logic program that terminates under the traditional logic programming semantics, trivially terminates under the tabled semantics. Since more programs and queries terminate under the tabled semantics, stronger methods for proving termination are needed. In the context of well-moded programs, Plümer [37] presents a sufficient condition for programs to have the bounded term-size property, which implies termination of tabled programs using left-to-right selection rule. Termination and mode analysis [41, 36] for non-tabled higher-order logic programs provides a starting point in our framework and can provide helpful clues on which predicates to table and which not. It seems interesting to strengthen already existing termination analysis for tabled higher-order logic programming to automatically decide what programs to table or at least guide the programmer in choosing tabling.

7.3 Tabled linear logic programming

In this proposal we focused on tabled higher-logic programming. Similar to the development of uniform proofs for a fragment of intuitionistic logic, we can design a higher-order logic programming language based on linear logic. Different proposals for linear logic programming languages exist such as Lolli [19] or LinLog [1] . Similar to the logical framework LF on which the language *Elf* is based, Cervesato and Pfenning [5] developed the linear logical framework and a linear logic programming language *LLF*, which is based on additive product, additive unit and linear and non-linear functions. Linear logic programming is ideally suited to describe planning problems, finite automata or process calculi such as the π-calculus. All linear logic programming interpretation are based on the traditional view. It seems plausible to extend the tabled logic programming paradigm to linear logic programming, at least for the additive fragment. As we often have a finite number of possible reachable states, the execution of these linear logic programs will terminate. Executing a linear logic program would then correspond to model-checking the program. The main practical impediment seems to be that computation in linear logic programming is essentially the transformation of a linear context Δ into another context Δ'. To achieve an efficient implementation, this requires the careful design of the table and indexing of linear contexts.

8 Evaluation

We have used the prototype as a logic programming interpreter for several examples from subtyping, intersection types, untyped lambda calculus and transition graphs. These logical specifications are not executable with the traditional logic programming interpreter. For the conducted experiments we were generally satisfied with the efficiency of the implementation, if the specifications had a finite model, i.e. there were finitely many answers to the query. If there are infinitely many answers to a query, then term and context depth suspension are critical to generate all answers in a fair manner. However, the size of the table and the number of suspended goals in all these examples did not exceed a few hundred table entries. It would be interesting to experiment with larger programs in this setting.

We also used the tabled logic programming interpreter to execute the implementations such as the refinement type-checker by Pfenning and Davies [9]. Non-tabled execution of the type-checker is severely hampered by redundant paths. The traditional logic programming interpreter will generate all possible paths to solve a type-check a given program, although we might only care about the existence of a proof that the program is type-correct. Using subsumption and strengthening, the refinement type checker executes some programs twice as fast as the original – this is without any sophisticated indexing data structures. When we execute the refinement type checker with the traditional logic programming interpreter, it generates the same solution 20736 times. For smaller examples, the depth-first interpreter outperforms the tabled logic interpreter. There are two main reasons for this: First, currently we table every predicate and every subgoal. This leads to a considerable overhead. Therefore it is critical to selectively table predicates and subgoals. Second, storing and retrieving goals and answers from the table does not use any sophisticated indexing data structures. Accessing and managing the table is a considerable problem if the table size exceeds a couple of hundred table entries. This becomes apparent when experimenting with some examples from Cartesian closed categories where the size of the table and the number of suspended goals increases dramatically and the current prototype cannot allocate enough memory.

The meta-theorem prover uses iterative deepening search, instead of the depth-first search employed by the logic programming interpreter. We can view the query as a theorem and then use the meta-theorem prover to find a proof for the query. In contrast to the logic programming interpreter, it will only generate one answer together with a proof, and not all possible answers. For small examples, the iterative deepening search and computation based on memoization behaved very similarly. However, specifications involving reflexivity, symmetry and transitivity cause a major problem for the iterative deepening search. An example is conversions in the specification of the untyped λ-calculus. While the performance of the theorem prover highly depends on the chosen clause orderings and on the size of the λ-terms, we found the tabled logic programming interpreter to be a more robust search procedure.

9 Conclusion

I am proposing tabled higher-order logic programming as a basis for efficiently executing logical systems and reasoning with and about them. The proof-theoretical characterization for tabled higher-order logic programming provides a high-level description for tabled search and forms the basis for applying this method also to other logic programming languages such as λProlog or linear logic programming. I intend to prove soundness of the tabled uniform search, and time permitting investigate completeness. Preliminary results with the implemented prototype seem promising. However to obtain an efficient tabled higher-order logic programming engine, the development of higher-order indexing data-structures and optimizations are critical. Experiments with the tabled search will provide greater insight into which optimizations are required and beneficial. Another major part of the work will be the integration of the tabled search into the meta-theorem prover. We expect that this will lead to a more robust and efficient meta-theorem prover.

References

1. Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.
2. 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.
3. David Basin and Toby Walsh. Difference matching. In *Proceedings of the 11th International Conference on Automated Deduction, Saratoga Springs, NY*, LNAI vol. 607, pages 295–309. Springer-Verlag, 1992.
4. Iliano Cervesato. Proof-theoretic foundation of compilation in logic programming languages. In J. Jaffar, editor, *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming — JICSLP'98*, pages 115–129, Manchester, UK, 16–19 June 1998. MIT Press.
5. Iliano Cervesato and Frank Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.
6. Iliano Cervesato and Frank Pfenning. Spine calculus. In *submitted*, 2001.

7. Weidong Chen, Michael Kifer, and David Scott Warren. HILOG: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.
8. Baoqiu Cui, Yifei Dong, Xiaoqun Du, K. Narayan Kumar, C.R. Ramakrishnan, I.V. Ramakrishnan, Abhik Roychoudhury, Scott A. Smolka, and David S. Warren. Logic programming and model checking. In Hugh Glaser Catuscia Palamidessi and Karl Meinke, editors, *Principles of Declarative Programming (Proceedings of PLILP/ALP'98)*, volume 1490 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 1998.
9. 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.
10. Steve Dawson, C. R. Ramakrishnan, Steve Skiena, and Terrance Swift. Principles and practice of unification factoring. *ACM Transactions on Programming Languages and Systems*, 18(6):528–563, 1995.
11. Steve Dawson, C. R. Ramakrishnan, and I. V.Ramakrishnan. Design and implementation of jump tables for fast indexing of logic programs. In *International Symposium on Programming Language Implementation and Logic Programming (PLILP'95)*, volume 982 of *Lecture Notes in Computer Science*, pages 133–150. Springer-Verlag, 1995.
12. Steven Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, and Terrance Swift. Optimizing clause resolution: Beyond unification factoring. In *International Logic Programming Symposium*, pages 194–208, 1995.
13. Stefaan Decorte, Danny De Schreye, Michael Leuschel, Bern Martens, and Konstantinos Sagonas. Termination analysis for tabled logic programming. In *Seventh International Workshop on Logic Program Synthesis and Transformation (LOPSTR'97), Leuven, Belgium, July 1997*, LNCS 1463, pages 111–127. Springer-Verlag, 1998.
14. Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, August 1993.
15. Harald Ganzinger, Robert Nieuwenhuis, and Pilar Nivela. Context trees. In *Prooceedings of the first International Joint Conference on Automated Reasoning, Siena, Italy*, LNAI 2083, pages 242–256. Springer-Verlag, 2001.
16. Peter Graf. Substitution tree indexing. In *Prooceedings of the sixth International Conference on Rewriting Techniques and Applications, Kaiserslautern, Germany*, LNCS 914, pages 117–131. Springer-Verlag, 1995.
17. Hai-Feng Guo, C.R. Ramakrishnan, and I.V. Ramakrishnan. Speculative beats conservative justification. In *International Conference on Logic Programming (ICLP'01)*, Nov 2001.
18. 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.
19. J. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. In *Proc. 6th Conf. on Logic in Computer Science*, pages 32–42, Amsterdam, 1991. IEEE Computer Society Press.
20. Jacob M. Howe. Two loop detection mechanisms: a comparison. In *Proceedings of the 6th Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, pages 188–200. Springer-Verlag, 1997. LNCAI 1227.
21. Lars Klein. Indexing für Terme höherer Stufe. Diplomarbeit, FB 14, Universität des Saarlandes, Saarbrücken, Germany, 1997.
22. Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358, 1992.
23. Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
24. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

25. Gopalan Nadathur and Dale Miller. An overview of λProlog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.
26. G. Necula and S. Rahul. Oracle-based checking of untrusted software. In *28th ACM Symposium on Principles of Programming Languages (POPL01)*, 2001.
27. George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 229–243, Berkeley, CA, USA, 1996. USENIX.
28. Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.
29. Lawrence C. Paulson. Isabelle: The next seven hundred theorem provers. In E. Lusk and R. Overbeek, editors, *Proceedings of the 9th International Conference on Automated Deduction*, pages 772–773, Argonne, Illinois, 1988. Springer Verlag LNCS 310. System abstract.
30. Frank Pfenning. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, June 1989. IEEE Computer Society Press.
31. Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.
32. Frank Pfenning. *Computation and Deduction*. Cambridge University Press, 2000. In preparation. Draft from April 1997 available electronically.
33. Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *16th Annual IEEE Symposium on Logic in Computer Science*, Boston, USA, 2001.
34. 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.
35. Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.
36. Brigitte Pientka. Termination and reduction checking for higher-order logic programs. In *Prooceedings of the first International Joint Conference on Automated Reasoning, Siena, Italy*, LNAI 2083, pages 401–415. Springer-Verlag, 2001.
37. Lutz Plümer. *Termination Proofs for Logic Programs*. LNAI 446. Springer-Verlag, 1990.
38. I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. Warren. Efficient access mechanisms for tabled logic programs. *Journal of Logic Programming*, 38(1):31–54, Jan 1999.
39. I. V. Ramakrishnan, R. Sekar, and A. Voronkov. Term indexing. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science Publishers B.V., 2001. To appear.
40. I.V Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. Warren. Efficient tabling mechanisms for logic programs. In L. .Sterling, editor, *Proceedings of the Twelth International Conference on Logic Programming*, pages 697–711. MIT Press, 1995.
41. Ekkehard Rohwedder and Frank Pfenning. Mode and termination checking for higher-order logic programs. In Hanne Riis Nielson, editor, *Proceedings of the European Symposium on Programming*, pages 296–310, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1058.
42. Abhik Roychoudhury, C. R. Ramakrishnan, and I. V. Ramakrishnan. Justifying proofs using memo tables. In *International Conference on Principles and Practice of Declarative Programming(PPDP'00)*, pages 178–189, 2000.

43. 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.
44. Carsten Schürmann. *Automating the meta theory of deductive systems.* PhD thesis, Department of Computer Sciences, Carnegie Mellon University, 2000. available as Technical Report CMU-CS-00-146.
45. Carsten Schürmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for LF. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction (CADE-15)*, pages 286–300, Lindau, Germany, July 1998. Springer-Verlag LNCS 1421.
46. R. Sekar, C.R. Ramakrishnan, I.V. Ramakrishnan, and Scott A. Smolka. Model-carrying code (MCC): A new paradigm for mobile-code security. In *New Security Paradigms Workshop (NSPW'01); Cloudcroft, New Mexico*, Sep 2001.
47. H. Tamaki and T. Sato. OLD resolution with tabulation. In E. .Shapiro, editor, *Proceedings of the 3rd International Conference on Logic Programming*, volume 225 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 1986.
48. T. Tammet. A resolution theorem prover for intuitionistic logic. In *Proceedings of the 13th International Conference on Automated Deduction, New Brunswick, NJ, July 1996*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 2–16, 1996.
49. Sophie Verbaeten, Konstantinos Sagonas, and Danny De Schreye. Modular termination proofs for Prolog with tabling. In *International Conference on Principles and Practice of Declarative Programming(PPDP'98)*, LNCS 1702, pages 342–359. Springer-Verlag, 1998.
50. Roberto Virga. *Higher-Order Rewriting with Dependent Types.* PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, 2000.
51. Toby Walsh. A divergence critic. In *Proceedings of the 12th International Conference on Automated Deduction, Nancy, France*, pages 14–28. Springer-Verlag, 1994.
52. David S. Warren. *Programming in tabled logic programming.* draft available from http://www.cs.sunysb.edu/ warren/xsbbook/book.html, 1999.
53. Christoph Weidenbach. Spass: Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning.* Elsevier, 2001. forthcoming.

10 Appendix

```
%% Declarations of types and expressions

tp: type.
zero : tp.
pos  : tp.
neg  : tp.
nat  : tp.
int  : tp.
=>   : tp -> tp -> tp.
%infix right 11 =>.
```

```
exp: type.
z    : exp.
s    : exp -> exp.
app  : exp -> exp -> exp.
lam  : (exp -> exp) -> exp.
letn : exp -> (exp -> exp) -> exp.
```

```
%% Subtyping specification
sub : tp -> tp -> type.
refl : sub T T.
tr   : sub T S
       <- sub T R
       <- sub R S.
zn   : sub zero nat.
pn   : sub pos nat.
nati : sub nat int.
negi : sub neg int.
arr  : sub (T1 => T2) (S1 => S2)
       <- sub S1 T1
       <- sub T2 S2.
```

```
%% Typing rules
of : exp -> tp -> type
tp_zz   : of z zero.
tp_zn   : of z nat.
tp_sp   : of (s E) pos
          <- of E nat.
tp_sn   : of (s E) nat
          <- of E nat.
tp_app  : of (app E1 E2) T
          <- of E1 (T2 => T)
          <- of E2 T2.
tp_lam  : of (lam ([x] E x)) (T1 => T2)
          <- ({x:exp} of x T1 -> of (E x) T2).
tp_letn : of (letn E1 ([x] E2 x)) T
          <- of E1 T1
          <- of (E2 E1) T.
tp_sub  : of E T
          <- of E T'
          <- sub T' T.
```