A proof-theoretic foundation for tabled higher-order logic programming

Brigitte Pientka*

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
bp@cs.cmu.edu

Abstract. Higher-order logic programming languages such as *Elf* extend first-order logic programming in two ways: first-order terms are replaced with (dependently) typed λ-terms and the body of clauses may contain implication and universal quantification. In this paper, we describe *tabled higher-order logic programming* where some redundant computation is eliminated by memoizing sub-computation and re-using its result later. This work extends Tamaki and Sato's search strategy based on memoization to the higher-order setting. We give a proof-theoretic characterization of tabling based on uniform proofs and prove soundness of the resulting interpreter. Based on it, we have implemented a prototype of a tabled logic programming interpreter for *Elf*.

1 Introduction

Tabled first-order logic programming has been successfully applied to solve complex problems such as implementing recognizers and parsers for grammars [25], representing transition systems CCS and writing model checkers [6]. The idea behind it is to eliminate redundant computation by memoizing sub-computation and re-using its results later. The resulting search procedure is complete and terminates for programs with the bounded-term size property. The XSB system [22], a tabled logic programming system, demonstrates impressively that tabled together with non-tabled programs can be executed efficiently.

Higher-order logic programming languages such as *Elf* [14] extend first-order logic programming in two ways: first-order terms are replaced with dependently typed λ-terms and the body of clauses may contain implication and universal quantification. It offers a generic framework for 1) implementing logical systems as *Elf* programs, 2) executing them and generating a certificate for each execution via an interpreter 3) checking certificates via type-checking and 4) reasoning with and about logical systems via a meta-level theorem prover *Twelf* [19]. One of its applications lies in "certifying code" where programs are equipped with a certificate (proof) that asserts certain safety properties. The safety policy can be represented as a higher-order logic program in *Elf*. Appel and Felty [1] use the

* This work was partially supported by NSF Grant CCR-9988281.

logic programming interpreter to execute the specification and generate a certificate that a given program fulfills a specified safety policy. Necula and Rahul [12] use a logic programming interpreter for checking the correctness of a certificate. In their case, the certificate is a bit-string that guides the logic programming interpreter to resolve non-deterministic choices. Representing and executing different safety policies using *Elf* reduces the effort required for each specific policies and offers an ideal environment for experimenting and combining safety policies. Proof search based on logic programs plays a central role in this setting, but redundant computation may hamper the performance and computation may not terminate, although the underlying domain is finite.

In this paper, we present *tabled higher-order logic programming* where some redundant computation is eliminated by memoizing sub-computation and re-using its result later. As higher-order logic programming allows nested implications and universal quantification in the body of clauses, goals might depend on a context of assumptions. We also have dependencies among terms, as the term language is derived from the dependently typed λ-calculus. The combination of both requires careful design of the table and table operations. We give a proof-theoretic characterization of tabled higher-order logic programming based on uniform proofs [10] and show soundness of the resulting interpreter. This work forms the basis of the implemented tabled interpreter for the language *Elf*. Although we concentrate on the logical framework LF, which is the basis of *Elf*, it seems possible to apply the presented approach to λProlog [11] or Isabelle [13], which are based on hereditary Harrop formulas and simply typed terms.

The paper is organized as follows: In Sec. 2 we introduce a type system for Mini-ML including subtyping. Using this example, we review briefly tabled logic programming and discuss higher-order tabled computation in Sec. 3. In Sec. 4 we review uniform proofs and then develop a tabled uniform proof system and prove soundness. In Sec. 5 we discuss related work and summarize the results.

2 A motivating example: subtyping

2.1 Background

As a running example we consider a type system for a restricted functional language Mini-ML, which includes subtyping. We only consider a small set of expressions, negative numbers $\mathsf{n}(e)$, natural numbers z and $\mathsf{s}(e)$, functions $\mathsf{lam}\ x.e$, function application $\mathsf{app}\ e_1\ e_2$. The type zero contains only the number z, the type pos represents all positive natural number and the type nat describes all natural numbers; the type neg denotes the negative numbers and the type int describes all numbers.

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

The specification of the subtyping relation using reflexivity and transitivity and the typing rules are straightforward (see Fig. 1). For a full description we refer the reader to [20].

$$\frac{}{\Gamma \vdash \mathsf{z} : \mathsf{zero}}\ \mathsf{tp\_zz} \qquad \frac{}{\Gamma \vdash \mathsf{n(z)} : \mathsf{neg}}\ \mathsf{tp\_negz}$$

$$\frac{\Gamma \vdash e : \mathsf{nat}}{\Gamma \vdash \mathsf{s}(e) : \mathsf{pos}}\ \mathsf{tp\_sp} \qquad \frac{\Gamma \vdash e : \mathsf{neg}}{\Gamma \vdash \mathsf{n}(e) : \mathsf{neg}}\ \mathsf{tp\_neg} \qquad \frac{\Gamma \vdash e : \tau' \qquad \tau' \preceq \tau}{\Gamma \vdash e : \tau}\ \mathsf{tp\_sub}$$

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

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

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

Fig. 1. Typing rules including subtyping relation

The subtyping relation is directly translated into *Elf* using logic programming notation. Constants `neg`, `zero`, `pos`, `nat` 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 $A_2 \leftarrow A_1$. From a logic programming point of view, it might be more intuitive to think of the clause $H \leftarrow A_1 \leftarrow \ldots \leftarrow A_n$ as $H \leftarrow A_1\ ,\ \ldots\ ,\ A_n$.

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

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

For implementing the subtyping relations logic programming based on Horn clauses suffices. However, *Elf* is much richer than first-order logic programming and also supports elegant encodings based on higher-order abstract syntax [18]. 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 $\mathsf{lam}\ x.e$ is represented as `lam [x] E x` in *Elf*. Substitution is modeled via application and β-reduction. In addition to the variable binding construct, *Elf* supports reasoning from hypotheses and handling parameters. The premise of typing rule for lam depends on the new parameter x and the hypothesis that x is of type τ_1. Moreover, we assume that it is possible to rename all variables in e, if necessary. In *Elf* this is represented by `({x:exp} of x`

`T1 -> of (E x) T2)` where `{x:exp}` denotes the universal quantifier $\Pi x\colon \text{exp}$. We can show `of (lam ([x] E x)) (T1 => T2)`, if we can prove that for a new variable `x`, if `x` has type `T1` then the body of the function `(E x)` has type `T2`. For a more detailed discussion see [16]

Higher-order logic programming suffers from the same problems as first-order logic programming: computation may be trapped in infinite paths and performance may be hampered by redundant computation. For example, the execution of the query `sub zero T` will end in an infinite branch trying to apply the transitivity rule. Similarly, the execution of the query `of (lam [x] x) T` will not terminate and fail to enumerate all possible types. In addition, we repeatedly type-check sub-expressions, which occur more than once. To eliminate redundancy, some sophisticated type checkers for example for refinement types memoize the result of sub-computations to obtain more efficient implementations. In this paper, we extend higher-order logic programming languages such as *Elf* with generic memoization techniques, called *tabled higher-order logic programming*. This has several advantages. Although it is possible to derive an algorithmic subtyping relation for the given example, this might not be trivial in general. To refine the implementation further by adding explicit support for memoization, complicates the type checker. As a consequence, the certificates, which are produced as a result of the execution, are larger and contain references to the explicit memoization data-structure. This is especially undesirable in the context of certified code where certificates are transmitted to and checked by a consumer, as sending larger certificates takes up more bandwidth and checking them takes more time. Moreover, proving the correctness of the type-checker with special memoization support will be hard, because we need to reason explicitly about the structure of memoization. As tabled logic programming terminates for programs with the bounded term-size property, we are also able to disprove certain statements. This in turn helps the user to debug the specification and implementations. In this paper, we propose to extend higher-order logic programming with memoization techniques.

2.2 Tabled logic programming: review

Tabling methods evaluate programs by maintaining tables of subgoals and their answers and by resolving repeated occurrences of subgoals against answers from the table. We review briefly Tamaki and Sato's multi-stage strategy [23], which differs only insignificantly from SLG resolution [5] for programs without negation. To demonstrate tabled computation, we consider the evaluation of the query $\mathsf{sub}\ \mathsf{zero}\ T$ in more detail.

The search proceeds in multiple stages. The table serves two purposes: 1) We record all sub-goals encountered during search. If the current goal is not in the table, then we add it to the table and proceed with the computation. Computation at a node is suspended, if the current goal is a variant of a table entry. 2) In addition to the sub-goals we are trying to solve, we also store the results of computation in the table as a list of answers to the sub-goals. To simplify the table in this presentation, we do not record the certificate (proof term) explicitly in the table, although we do record it in the actual implementation. In each stage we apply program clauses and answers from the table. Figure 2 illustrates the search process.

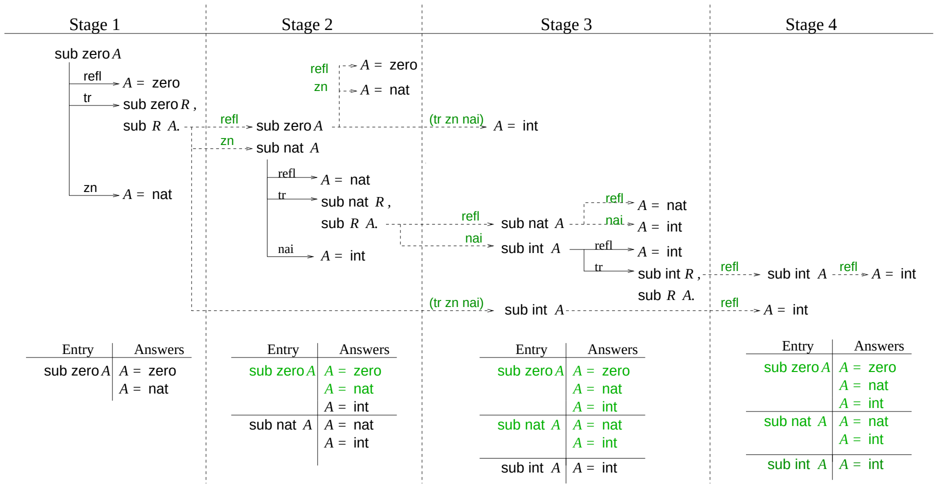

Fig. 2. Staged computation

The root of the search tree is labeled with the goal sub zero A. Each node is labeled with a goal statement and each child node is the result of applying a program clause or an answer from the table to the *leftmost* atom of the parent node. Applying a clause $H \leftarrow A_1 \leftarrow A_2 \ldots \leftarrow A_n$ results in the subgoals $A_1, A_2, \ldots, A_n$ where all of these subgoals need to be satisfied. We will then expand the first subgoal A_1 carrying the rest of the subgoals $A_2, \ldots, A_n$ along. If a branch is successfully solved, we show the obtained answer. To distinguish between program clause resolution and re-using of answers, we have two different kinds of edges in the tree. The edges obtained by program clause resolution are solid while edges obtained by reusing answers from the table are dashed. Both are labeled with the clause name that was used to derive the child node. Using the labels at the edges we can reconstruct the proof term for a given query. In general, we will omit the actual substitution under that the parent node unifies with the program clause to avoid cluttering the example. To ensure we generate all possible answers for the query, we restrict the re-use of answers from the table. In each stage, we are only allowed to re-use answers that were generated in previous stages. Answers from previous stages (available for answer resolution) are marked gray, while current answers (not available yet) are black.

3 Tabled higher-order logic programming

In tabled higher-order logic programming, we extend tabling to handle subgoals that may contain implications and universal quantification and our term language is the dependently typed λ-calculus. The table entries are no longer atomic goals, but atomic goals A together with a context Γ of assumptions. In addition, terms might depend on assumptions on Γ. To highlight some of the challenges we present the evaluation of the query of (lam $[x]$ x) T in Fig. 3.

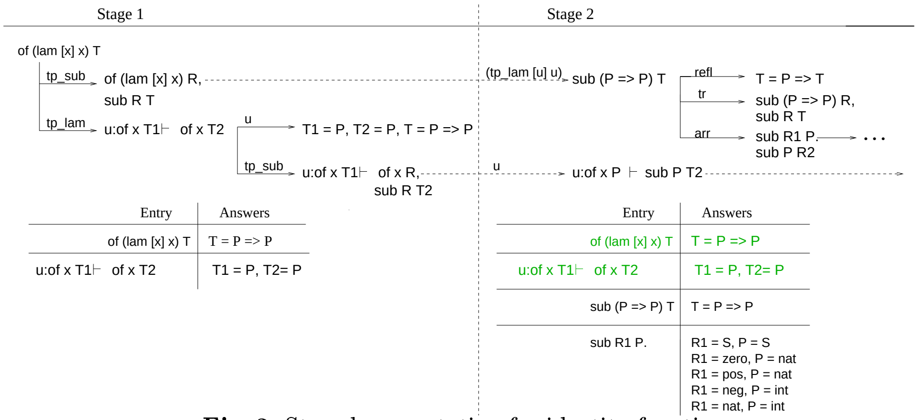

Fig. 3. Staged computation for identity function

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

Due to the higher-order setting, the predicates and terms might depend on Γ. Virga [24] developed in his PhD thesis techniques, called subordination, to analyze dependencies in *Elf* programs statically before execution. In the Mini-ML example, the terms of type `exp` and `tp` are independent of each other. On the level of predicates, the type checker `of` depends on the subtyping relation `sub`, but not vice versa. When checking whether a subgoal $\Gamma \vdash a$ is already in the table, we exploit the subordination information in two ways. First, we use it to analyze the context Γ and determine which assumptions might contribute to the proof of a. For example the proof for u:of x $T_1 \vdash$ of x T_2 depends on the assumption u. However, the proof for u:of x $P \vdash$ sub P T_2 cannot depend on the assumption u, as the predicate sub does not refer to the predicate of . Therefore, when checking whether u:of x $P \vdash$ sub P T_2 is already in the table, it suffices to look for a variant of $\mathsf{sub}\ P\ T_2$. In the given example, computation at subgoal $u{:}\mathsf{of}\ x\ P \vdash \mathsf{sub}\ P\ T_2$ is suspended during stage 2 as the table already contains $\mathsf{sub}\ R_1\ P$. If we for example first discover $u{:}\mathsf{of}\ x\ P \vdash \mathsf{sub}\ P\ T_2$, then we store the strengthened goal $\mathsf{sub}\ P\ T_2$ in the table with an empty context.

Second, subordination provides information about terms. As we are working in a higher-order setting, solutions to new existential variables, which are introduced during execution, might depend on assumptions from Γ. For example, applying the subtyping rule to $u{:}\mathsf{of}\ x\ T_1 \vdash \mathsf{of}\ x\ T_2$ yields the new goal $u{:}\mathsf{of}\ x\ T_1 \vdash \mathsf{of}\ x\ (R\ x\ u)\ ,\ \mathsf{sub}\ (R\ x\ u)\ T_2$ where the solution for the new variable R might depend on the new variable $x{:}\exp$ and the assumptions $u{:}\mathsf{of}\ x\ T_1$. However, we know that the solution must be an object of `tp` and that objects of `tp` are independent of Mini-ML expressions `exp` and the Mini-ML typing rules `of`. Hence, we can omit x and u and write $u{:}\mathsf{of}\ x\ T_1 \vdash \mathsf{of}\ x\ R,\ \mathsf{sub}\ R\ T_2$. Before comparing goals with table entries and adding new table entries, we eliminate unnecessary dependencies from the subgoal $\Gamma \vdash a$. This allows us to detect more loops in the search tree and eliminate more redundant computation. For further discussion issues in higher-order tabling, we refer the interested reader to [20].

4 A foundation for tabled higher-order logic programming

4.1 Uniform proofs

Computation in logic programming is achieved through proof search. Given a goal (or query) A and a program Γ, we derive A by successive application of clauses of the program Γ. Miller *et al* [10] 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.*

Elf is one example of an abstract logic programming language, which is based on the LF type theory. Π-quantifier and $\rightarrow$ suffice to describe LF. 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;

Types $A ::= a \mid A_1 \rightarrow A_2 \mid \Pi x : A_1.A_2$	Terms $M ::= H \cdot S \mid \lambda x : A.M$
Programs $\Gamma ::= \cdot \mid \Gamma, x : A$	Spines $S ::= \textsc{nil} \mid M; S$
	Heads $H ::= c \mid x$

a ranges over atomic formulas. The function type $A_1 \rightarrow A_2$ corresponds to an implication. The Π-quantifier, denoting dependent function type, can be interpreted as the universal $\forall$-quantifier. The clause `tr:sub T S <- sub T R <-`

`sub R S.` is interpreted as $\mathsf{tr}{:}\Pi t{:}\mathsf{tp}.\Pi s{:}\mathsf{tp}.\Pi r{:}\mathsf{tp}.\ \mathsf{sub}\ r\ s \to (\mathsf{sub}\ t\ r \to \mathsf{sub}\ t\ s)$. Every type has a corresponding proof term M and we assume that all proof terms are in normal form. In the example from Sec. 2, the proof term corresponding to sub zero int is given as tr zn refl. Note that we actually omitted the implicit arguments zero and nat, which denote the instantiation of transitivity rule. In the following discussion, we will include implicit arguments in the proof term representation. To represent proof terms, we use the spine notation [4]. To give an intuition for this notation we give a few examples. tr zero nat nat zn refl is denoted using spine notation by $\mathsf{tr} \cdot \mathsf{zero}\ ;\ \mathsf{nat}\ ;\ \mathsf{nat}\ ;\ \mathsf{zn}\ ;\ \mathsf{refl}\ ;\ \textsc{nil}$. The proof term $\mathsf{tp\_lam}\ T\ (\lambda x{:}\exp.x)\ T\ (\lambda x{:}\exp.\lambda u{:}\mathsf{of}\ x\ T.\ u)$ is denoted by $\mathsf{tp\_lam} \cdot T\ ;\ (\lambda x{:}\exp.x \cdot \textsc{nil})\ ;\ T\ ;\ (\lambda x{:}\exp.\lambda u{:}\mathsf{of}\ x\ T.\ u \cdot \textsc{nil})\ ;\ \textsc{nil}$.

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

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

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

Fig. 4. Uniform deduction system for $\mathcal{L}$

We can characterize uniform proofs by two main judgments: $\Gamma \xrightarrow{u} M : A$ says there is a uniform proof M for A from the program Γ and $\Gamma \gg A \xrightarrow{f} S : a$ there exists a focused proof S for the atom a by focusing on program clause A. Taking a type-theoretic view, we can interpret the first judgment as M has type A in the context Γ and the later as S has type a in context Γ. Inference rules describing uniform and focused proofs are given in Fig. 4. 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 [9] 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 $\Pi x_1{:}B_1.\dots\Pi x_n{:}B_n.A_1$ where Γ is a context $x_1{:}B_1, \dots, x_n{:}B_n$ and $M' \cdot \Gamma$ as an abbreviation for $M' \cdot x_1; \dots ; x_n; \textsc{nil}$. Moreover, there is a one-to-one correspondence between these two solutions, as $M' \cdot \Gamma$ reduces to M. Following Miller's terminology, we say M' is the result of raising M. Pfenning [15] investigated this notion in the setting of the calculus of construction, which includes LF. Intuitively, M depends globally on the assumptions in Γ. Raising allows us to localize dependencies by replacing M with $M' \cdot \Gamma$. This step becomes important when we want to compute the instantiation of x in the f$\forall$ rule by introducing a existential variable X and instantiating X by unification. We come back to it in the next section. To reflect raising, we rewrite the $\mathsf{f}\forall$ rule to:

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

The proof term represents the witness of the proof. When searching for a uniform proof, the proof term is constructed simultaneously. 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.

4.2 Uniform proofs with answer substitutions

The result of a computation in logic programming is generally an answer substitution θ for the existentially quantified variables in a goal A. To obtain an algorithm that computes answer substitutions, we substitute existential variables X for the bound variable x in the $\mathsf{f}\forall$ rule. In fact, we replace M' with X in the raised version of $\mathsf{f}\forall$ rule. Note that X does not globally depend on Γ anymore as raising allowed us to rotate the existentially quantified variables to the outside. As all dependencies are local, we can model dependencies between parameters and existential variables by annotating existential variables X with their type. Existential variables are instantiated later during unification yielding a substitution θ. An alternative would be to use mixed-prefixes [9] 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[15]. Many programs fall into the decidable fragment and we concentrate on this case.

$$\begin{array}{ll} \text{Substitution } \theta ::= \cdot \mid \theta, X_A = M & \\ \text{Composition} & \cdot \circ \theta = \theta \\ & (\theta_1, X_A = M) \circ \theta_2 = (\theta_1 \circ \theta_2), X_{A[\theta_2]} = M[\theta_2] \end{array}$$

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 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. 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 $\theta \circ \theta = \theta$. 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 $\Gamma \xrightarrow{u} A/\theta$ and $\Gamma \gg A \xrightarrow{f} a/\theta$.

The inference rules are given in Fig. 5. To obtain an algorithm, we impose left-to-right order on the solution of the $\mathsf{fs} \rightarrow$ rule. This matches our intuitive understanding of computation in logic programming. In the $\mathsf{fs} \rightarrow$ 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 $\mathsf{fs}\forall$ rule we delay the instantiation of x by introducing a new existential variable X. In the $\mathsf{fs\_atom}$ 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 $\Gamma \vdash a \doteq a'$ where a and a' are higher-order patterns.

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

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

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

Fig. 5. Uniform deduction system for $\mathcal{L}_\theta$ with substitutions

There is still some non-determinism left in $\mathcal{L}_\theta$, which needs to be resolved in an actual implementation. In the $\mathsf{us\_atom}$ rule, we do not specify which clause from Γ we pick and focus on. Logic programming interpreter usually try the clauses in the order they occur as backtracking over different choices is considered too expensive. This choice renders the search strategy incomplete in practice. However, the presented deductive system $\mathcal{L}_\theta$, which generates answer substitutions, is sound and complete, as expressed by the following two theorems.

Theorem 1 (Soundness). *If $\mathcal{D} : \Gamma \xrightarrow{u} A/\theta$ then for any grounding substitution σ' which grounds $\Gamma \xrightarrow{u} A/\theta$ we have $\mathcal{E}{:}\Gamma[\theta \circ \sigma'] \xrightarrow{u} A[\theta \circ \sigma']$.*

Theorem 2 (Completeness). *If $\mathcal{E} : \Gamma \xrightarrow{u} A[\sigma]$ for a grounding substitution σ then $\mathcal{D} : \Gamma \xrightarrow{u} A/\theta$ for some θ and $\sigma = \theta \circ \sigma'$ for some σ'.*

The proof requires a more general statement that also includes focused proofs, but otherwise is straightforward. In the next section, we extend this system $\mathcal{L}_\theta$ to include memoization.

4.3 Tabled uniform proofs

The idea behind tabled uniform proofs 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. We keep in mind that we are silently generating proof terms together with answer substitution. We assume that some predicates are designated as tabled predicates where we record subgoals and corresponding answers. For predicates not designated as tabled the $\mathsf{us\_atom}$ rule still applies.

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 pairs, answer substitutions θ and proof terms M, such that $\Gamma[\theta] \xrightarrow{u} M[\theta] : a[\theta]$ is a solution.*

The table is a store of proven and still open conjectures. The open conjectures are the table entries that have an empty list of answers. The proven conjectures (lemmas) are the table entries that have a list of answer substitutions associated with them. As proof terms are generated and stored together with answer substitutions, we also have the actual proof for the given conjecture. We will design the inference rules in such a way that for any solution in the table $\Gamma[\theta] \xrightarrow{u} M[\theta] : a[\theta]$ there exists a derivation $\Gamma \xrightarrow{u} M : a/\theta$. We will keep all the previous inference rules, but keep in mind that we are silently passing around a table $\mathcal{T}$. Any substitution we apply to Γ and a (see for example the $\mathsf{fs} \rightarrow$ 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. The main judgments are $\mathcal{T}; \Gamma \xrightarrow{u} M : A/(\theta, \mathcal{T}')$ and $\mathcal{T}; \Gamma \gg A \xrightarrow{f} S : a/(\theta, \mathcal{T}')$.

In addition to the $\mathsf{us\_atom}$ inference rule, we will have the rules extend and $\mathsf{retrieve}$. The extend rule adds a subgoal and its answer to the table. When we encounter a new subgoal, we add a new entry with an empty answer list to the table. Once we have proven this subgoal, we add the answer substitution and proof term to its answer list, and we can later use it as a lemma. $\mathsf{retrieve}$ allows us to close a branch by applying a lemma from the table. If we are proving $\Gamma \xrightarrow{u} a$, where Γ and a may contain existential variables and we have a proof for $\Gamma[\theta] \xrightarrow{u} M[\theta] : a[\theta]$ in the table then we can just re-use it by substituting the proof term $M[\theta]$for it. Applying the $\mathsf{retrieve}$ rule corresponds to introducing a cut in the proof using a lemma from the table. However, this cut application

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

Fig. 6. Memoization extensions

is restricted to using only lemmas that are an instance of the sequent we are trying to prove.

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 free 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.

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 $\mathcal{T}$ and $\mathcal{A}$ is non-empty such that $\Gamma' \xrightarrow{u} a'$ is a variant of $\Gamma \xrightarrow{u} a$ 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 renaming 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*$(\mathcal{T}, \Gamma \xrightarrow{u} a, M, \theta) = \mathcal{T}'$
Let $\mathcal{T}$ be a table, $\Gamma \xrightarrow{u} a$ be a goal and θ be a corresponding answer substitution and M the proof term. 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 obtain the substitution θ' where $\theta' = \sigma \circ \theta$. Then we add $(\theta', M[\sigma])$ 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, which 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 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 which are not already in the table yet, we need to still be able to apply extend rule.

Definition 5 (retrieve). $\mathsf{retrieve}(\mathcal{T}, \Gamma \xrightarrow{u} a) = (\theta, M)$
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* $\Gamma \xrightarrow{u} a$ *and* (θ_i, M) *is in* $\mathcal{A}_i$ *then match* $\Gamma \xrightarrow{u} a$ *against* $\Gamma_i[\theta_i] \xrightarrow{u} a_i[\theta_i]$ *to obtain a substitution* θ *where* $\theta_i = \sigma \circ \theta$ *and return* θ *and* $M = M_i[\sigma]$.

Theorem 3 (Soundness). *If* $\mathcal{D} : \mathcal{T}; \Gamma \xrightarrow{u} A/(\theta, \mathcal{T}')$ *then for any substitution* σ *which grounds* $\mathcal{T}; \Gamma \xrightarrow{u} A/(\theta, \mathcal{T}')$ *we have* $\mathcal{E} : \Gamma \xrightarrow{u} A/\theta \circ \sigma$.

The proof requires again a generalization to include focused proofs, but is otherwise straightforward.

The presented inference rules leave several choices undetermined. For example, we do not specify the order in which we use program clauses. This choice was already present in the non-tabled system. Similarly, the rules to allow memoization leave open in what order we retrieve answers, when to retrieve answers and when to apply program clauses. Although we used variant checking in the definitions, it is possible to allow subsumption checking. In an actual implementation all these choices need to be resolved. The multi-stage strategy discussed earlier is one possible solution. In this strategy we proceed in lock-steps. First, we apply the extend rule until all clauses from Γ have been tried, and then allow the application of the retrieve rule. The strategy also restricts the retrieve rule, i.e. only answers from previous stages can be retrieved. Alternatively, we could use SCC scheduling (strongly connected components), which allows us to consume answers as soon as they are available [22]. In a real implementation we want to suspend goals and be able to later resume them. For example, 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. Storing suspended avoids repeating partial 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.

5 Related Work and Conclusion

This proof-theoretic view on computation based on memoization provides a high-level description of a tabled logic programming interpreter and separates logical issues from procedural ones leaving maximum freedom to choose particular control mechanisms. In fact, it is very close to our prototype implementation for *Elf*. So far all descriptions of tabling are highly procedural, either designed as an extension of SLD resolution [23] or to the WAM abstract machine[22]. Certificates, which provide evidence for the existence of a proof, have been added to tabled logic programming by Roychoudhury [21] and are called justifiers. The relationship between the certificate and SLD resolution is extrinsic rather then intrinsic and needs to be established separately. The proof-theoretical characterization offers a uniform framework for describing and reasoning about program clauses, goals and certificates (proof terms). It seems possible to apply the techniques described to other logic programming languages such as λProlog. Linear logic programming [8, 2] has been proposed as an extension of higher-order logic programming to model imperative state changes in a declarative (logical) way. We believe our techniques can be extended to cover this case, but it requires some new considerations. In particular, we plan to investigate the interaction between resource management strategies [3] or constraints [7] with tabling.

With tabled uniform proof search we will find fewer proofs than with uniform proofs. For example in the subtyping example given in Sec. 2 the query sub zero zero has infinitely many proofs under the traditional logic programming interpretation while we find only one proof under the tabled 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 [17] 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 if they produce the same answer substitution. In this setting, it seems possible to show that search based on tabled uniform proofs is also non-deterministically complete, i.e. if computation fails, then there exists no proof.

We have implemented the search strategy based on memoization for *Elf*. It not only allows us to execute more specifications, but also execute implementations more efficiently. Preliminary experiments include type checking for subtyping and intersection types, parsing into higher-order abstract syntax, evaluation based on rewriting. The most pressing issue seems to be to implement indexing data-structures to reduce the overhead involved in managing the table[1].

Acknowledgment: The author gratefully acknowledges numerous discussion with Frank Pfenning and David S.Warren concerning this work. Thanks also for many useful comments from C. Schürmann, R. Harper and K. Watkins.

References

1. W. Appel and A. 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, 2000.
2. I. Cervesato. *A Linear Logical Framework*. PhD thesis, Dipartimento di Informatica, Università di Torino, 1996.
3. I. Cervesato, J. S. Hodas, and F. Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232(1–2):133–163, 2000.
4. I. Cervesato and F. Pfenning. A linear spine calculus. In *submitted*, 2001.
5. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.
6. B. Cui, Y. Dong, X. Du, K. N. Kumar, C.R. Ramakrishnan, I.V. Ramakrishnan, A. Roychoudhury, S.A. Smolka, and D.S. Warren. Logic programming and model checking. In *Principles of Declarative Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 1998.
7. J. Harland and D. Pym. Resource-distribution via boolean constraints. In *Proceedings of the 14th International Conference on Automated Deduction (CADE-14)*, pages 222–236, Townsville, Australia, 1997. Springer-Verlag LNAI 1249.

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

8. J. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.
9. D. Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358, 1992.
10. D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
11. G. Nadathur and D. Miller. An overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, 1988. MIT Press.
12. G. Necula and S. Rahul. Oracle-based checking of untrusted software. In *28th ACM Symposium on Principles of Programming Languages (POPL01)*, 2001.
13. L. C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.
14. F. 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, 1989. IEEE Computer Society Press.
15. F. Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, Netherlands, 1991.
16. F. Pfenning. *Computation and Deduction.* Cambridge University Press, 2000. In preparation. Draft from April 1997 available electronically.
17. F. Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *16th Annual IEEE Symposium on Logic in Computer Science*, Boston, USA, 2001.
18. F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, 1988.
19. F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, 1999. Springer-Verlag LNAI 1632.
20. B. Pientka. Tabled higher-order logic programming. Thesis proposal, Carnegie Mellon University, 2002.
21. A. 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.
22. K. Sagonas and T. 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.
23. 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.
24. R. Virga. *Higher-Order Rewriting with Dependent Types.* PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, 2000.
25. D. S. Warren. *Programming in tabled logic programming.* draft available from http://www.cs.sunysb.edu/ warren/xsbbook/book.html, 1999.