Tabling for higher-order logic programming

Brigitte Pientka

School of Computer Science
McGill University
bp@cs.mcgill.ca

Abstract. We describe the design and implementation of a higher-order tabled logic programming interpreter where some redundant and infinite computation is eliminated by memoizing sub-computation and re-using its result later. In particular, we focus on the table design and table access in the higher-order setting where many common operations are undecidable in general. To achieve a space and time efficient implementation, we rely on substitution factoring and higher-order substitution tree indexing. Experimental results from a wide range of examples (propositional theorem proving, parsing, refinement type checking, small-step evaluator) demonstrate that higher-order tabled logic programming yields a more robust and more efficient proof procedure.

1 Introduction

Efficient redundancy elimination techniques such as loop detection or tabling play an important role in the success of first-order theorem proving and logic programming systems. The central idea of tabling is to eliminate infinite and redundant computation by memoizing subcomputation and reusing its results later on. Up to now, higher-order theorem proving and logic programming systems lack such memoization techniques, thereby limiting their success in many applications. This paper describes the design and implementation of tabling for the higher-order logic programming systems Twelf [16, 18] and presents a broad experimental evaluation demonstrating the feasibility and benefits of tabling in the higher-order setting.

Higher-order logic programming as Twelf [16] or λProlog [12] extends first-order logic programming along two orthogonal dimensions: First, we allow dynamic assumptions to be added and used during proof search. Second, we allow a higher-order term language which contains terms defined via λ-abstraction. Moreover, execution of a query will not only produce a yes or no answer, but produce a proof term as a certificate which can be checked independently. These features make higher-order logic programming an ideal generic framework for implementing formal systems and executing them.

Most recently, higher-order logic programming has been successfully employed in several certified code projects, where programs are equipped with a certificate (proof) that asserts certain safety properties [3, 5, 2]. The safety policy can be represented as a higher-order logic program and the higher-order logic

programming interpreter can be used to execute the specification and generate a certificate that a given program fulfills a specified safety policy. However, these applications also demonstrate that present technology is inadequate to permit prototyping and experimenting with safety and security policies. Many specifications are not directly executable and long response times lead to slow development of safety policies in many applications. In [19], we outline a proof-theoretic foundation for tabled proof search to overcome some of these deficiencies, by memoizing sub-computations and re-using its result later. This paper focuses on the realization and implementation of a tabled higher-order logic programming interpreter in practice and presents a broad experimental evaluation. This work is inspired by the success of memoization techniques in tabled first-order logic programming, namely the XSB system [24] where it has been applied in different problem domains such as implementing recognizers for grammars [27], representing transition systems CCS, writing model checkers [6].

In the higher-order setting, tabling introduces several complications. First, we must store intermediate goals together with dynamic assumptions which may be introduced during proof search. Second, many operations necessary to achieve efficient table access such unifiability or instance checking, are undecidable in general for higher-order terms. Our approach relies on linear higher-order patterns[21] and adapts higher-order substitution tree indexing [19] to permit lookup and possible insertion of terms to be performed in a single pass. To avoid repeatedly scanning terms when reusing answers, we adapt substitution factoring[22]. Third, since storing and reusing fully explicit proof terms to certify tabled proofs is impractical due to their large size, we propose a compact proof witness representation inspired by [13] which only keeps track of a proof footprint. As the experimental results from a wide range of examples (propositional theorem proving, refinement type checking, parsing, small-step evaluation) demonstrate, tabling leads to a more robust and more powerful higher-order proof search procedure.

The paper is organized as follows: In Sec. 2 we introduce higher-order logic programming using an example from propositional theorem proving. In Sec. 3 we describe the basic principles behind table design guided by substitution factoring and linearization of higher-order terms. This is followed by higher-order term indexing (Sec. 4), and compact proof witness generation (Sec. 5). Experimental results are discussed in Sec. 6. In Sec. 7 we discuss related work and summarize the results.

2 Motivating example:Sequent calculus

To illustrate the proof search problems and challenges in higher-order logic programming, we introduce a sequent calculus which includes implication, conjunction, and universal quantification. This logic can be viewed as a simple example of a general safety logic. It is small, but expressive enough that it allows us to discuss the basic principles and challenges of proof search in this setting. It can also easily be extended to a richer fragment which includes the existential quantifier, disjunction and falsehood. We will focus here on the higher-order logic program-

ming language Elf [16], which is based on the logical framework LF[9]. We will briefly discuss the representation of a first-order logic in the logical framework LF, and then illustrate how higher-order logic programming interpreter proceeds and what problems arise. We can characterize this fragment of first-order logic as follows:

$$\begin{array}{lll} \text{Propositions } A, B, C & := & P \mid \mathsf{true} \mid A \wedge B \mid A \supset B \mid \forall x.A \\ \text{Context} \qquad \Gamma & := & . \mid \Gamma, A \end{array}$$

The main judgment to describe provability is: $\Gamma \Longrightarrow A$ which means proposition A is provable from the assumptions in Γ. The rules for the intuitionist sequent calculus are straightforward.

$$\frac{\Gamma \Longrightarrow A \quad \Gamma \Longrightarrow B}{\Gamma \Longrightarrow A \wedge B}\ \mathsf{andR} \qquad \frac{\Gamma, A \wedge B, A \Longrightarrow C}{\Gamma, A \wedge B \Longrightarrow C}\ \mathsf{andL}_1 \qquad \frac{\Gamma, A \wedge B, B \Longrightarrow C}{\Gamma, A \wedge B \Longrightarrow C}\ \mathsf{andL}_2$$

$$\frac{\Gamma, A \Longrightarrow B}{\Gamma \Longrightarrow A \supset B}\ \mathsf{impR} \qquad \frac{\Gamma, A \supset B \Longrightarrow A \quad \Gamma, A \supset B, B \Longrightarrow C}{\Gamma, A \supset B \Longrightarrow C}\ \mathsf{impL}$$

$$\frac{}{\Gamma, A \Longrightarrow A}\ \mathsf{axiom} \qquad \frac{\Gamma \Longrightarrow [a/x]A \quad a \text{ is new}}{\Gamma \Longrightarrow \forall x.A}\ \mathsf{allR} \qquad \frac{\Gamma, \forall x.A, [T/x]A \Longrightarrow C}{\Gamma, \forall x.A \Longrightarrow C}\ \mathsf{allL}$$

The logical framework LF is ideally suited to support the representation and implementation of logical systems such as the intuitionist sequent calculus above. The representation of formulas and judgments follows [9]. We will distinguish between propositions ($\mathsf{conc}\ A$) we need to prove and propositions ($\mathsf{hyp}\ A$) we assume. The main judgment to show that a proposition A is provable from the assumptions $A_1, \ldots, A_n$ can be then viewed as: $\mathsf{hyp}\ A_1, \ldots, \mathsf{hyp}\ A_n \Longrightarrow \mathsf{conc}\ A$

This will allow a direct representation within the logical framework LF and the higher-order logic program describing the inference rules is given next.

$$\begin{array}{ll}
\mathsf{axiom} : \mathsf{conc}\ A & \mathsf{impR} : \mathsf{conc}\ (A\ \mathsf{imp}\ B) \\
\quad \leftarrow \mathsf{hyp}\ A. & \quad \leftarrow (\mathsf{hyp}\ A \rightarrow \mathsf{conc}\ B). \\
\mathsf{andR} : \mathsf{conc}\ (A\ \mathsf{and}\ B) & \mathsf{impL} : \mathsf{conc}\ C \\
\quad \leftarrow \mathsf{conc}\ A & \quad \leftarrow \mathsf{hyp}\ (A\ \mathsf{imp}\ B) \\
\quad \leftarrow \mathsf{conc}\ B. & \quad \leftarrow \mathsf{conc}\ A \\
\mathsf{andL}_1 : \mathsf{conc}\ C & \quad \leftarrow (\mathsf{hyp}\ B \rightarrow \mathsf{conc}\ C). \\
\quad \leftarrow \mathsf{hyp}\ (A\ \mathsf{and}\ B) & \mathsf{allR} : \mathsf{conc}\ (\mathsf{forall}\ \lambda x.A\ x) \\
\quad \leftarrow (\mathsf{hyp}\ A \rightarrow \mathsf{conc}\ C). & \quad \leftarrow \Pi x{:}\mathsf{i}.\mathsf{conc}\ (A\ x). \\
\mathsf{andL}_2 : \mathsf{conc}\ C & \mathsf{allL} : \mathsf{conc}\ C \\
\quad \leftarrow \mathsf{hyp}\ (A\ \mathsf{and}\ B) & \quad \leftarrow \mathsf{hyp}\ (\mathsf{forall}\ \lambda x.A\ x) \\
\quad \leftarrow (\mathsf{hyp}\ B \rightarrow \mathsf{conc}\ C). & \quad \leftarrow (\mathsf{hyp}\ (A\ T) \rightarrow \mathsf{conc}\ C).
\end{array}$$

There are two key ideas which make the encoding of the sequent calculus elegant and direct. First, we use higher-order abstract syntax to encode the bound variables in the universal quantifier. We can read the allR clause as follows: To prove $\mathsf{conc}\ (\mathsf{forall}\ \lambda x.A\ x)$ we need to prove for all parameters x, that

conc $(A\ x)$ is true, where the Π-quantifier denotes the universal quantifier in the meta-language. Second, we use the power of dynamic assumptions which higher-order logic programming provides, to eliminate the need to manage assumptions in a list explicitly. To illustrate, we consider the clause impR. To prove conc $(A$ imp $B)$, we prove conc B assuming hyp A. In other words, the proof for conc B may use the dynamic assumption hyp A.

When we need to access an assumption from the context, we simply try to prove hyp A using the axiom clause axiom. All the left rules andL_1, andL_2, impL and allL follow the same pattern. The andL_1 rule can be read operationally as follows: we can prove conc C, if we have an assumption hyp $(A$ and $B)$ and we can prove conc C under the assumption hyp A.

For the propositional fragment of the sequent calculus, proof search is decidable. Therefore, we expect that simple examples such as conc $((A$ and $B)$ imp $B)$ should be easily be provable. Unfortunately, an execution with a depth-first search interpreter will lead to an infinite loop, as we will continue to apply the andL_1 rule, and generate the following subgoals.

Dynamic assumption	Goal	Justification
A:o, B:o	$\vdash$ conc $((A$ and $B)$ imp $B)$	
A:o, B:o, h_1:hyp $(A$ and $B)$	$\vdash$ conc B	impR
A:o, B:o, h_1:hyp $(A$ and $B)$, h_2:hyp A	$\vdash$ conc B	andL_1
A:o, B:o, h_1:hyp $(A$ and $B)$, h_2:hyp A, h_3:hyp A	$\vdash$ conc B	loop

To prevent looping, we need to detect two independent problems. First, we need to prevent adding dynamic assumptions, which are already present in some form. However, this is only part of the solution, since we also need to detect that we keep trying to prove the goal conc B. In this paper, we will propose the use of tabling in higher-order logic programming to detect loops. The essential idea is to memoize subgoals together with its dynamic assumptions and re-use their results later. This will prevent that the computation will be trapped in infinite paths and can potentially improve performance by re-using the result of previous proofs. Note that although the subgoals encountered in the previous example were all ground, and did not contain any existential variables, this may in general not be the case. Consider for the slightly different version of the previous example which corresponds to $\exists y'.\forall x.\exists y.((Q\ y') \wedge (P\ x)) \supset (P\ y))$:

$$\text{exists } \lambda y'.\ \text{forall } \lambda x\ \text{exists } \lambda y\ (((Q\ y')\ \text{and}\ (P\ x))\ \text{imp}\ \ (P\ y))$$

We first remove the existential quantifier by introducing an existential variable Y'. Next, we eliminate the allR-rule by introducing a new parameter x. Then we remove the second existential quantifier, by introducing a second existential variable Y. Existential variables (or logic variables) such as Y' and Y are subject to higher-order unification during proof search. Parameter dependencies such as that the existential Y is allowed to depend on the parameter x, while Y' is not, is naturally enforced by allowing higher-order terms and existential variables

which can be instantiated with functions. Using the impR-rule, we introduce the assumption $\mathsf{hyp}\ ((Q\ Y')\ \mathsf{and}\ (P\ x))$, and the sequence of subgoals we will then encounter by continuing to apply the andL_1-rule is:

$$\begin{array}{ll}
x{:}\mathsf{i}, u{:}\mathsf{hyp}\ ((Q\ Y')\ \mathsf{and}\ (P\ x)) & \vdash \mathsf{conc}\ (P\ (Y\ x)) \\
x{:}\mathsf{i}, u{:}\mathsf{hyp}\ ((Q\ Y')\ \mathsf{and}\ (P\ x)), u_1{:}\mathsf{hyp}\ (Q\ Y') & \vdash \mathsf{conc}\ (P\ (Y\ x))\ \mathsf{andL}_1 \\
x{:}\mathsf{i}, u{:}\mathsf{hyp}\ ((Q\ Y')\ \mathsf{and}\ (P\ x)), u_1{:}\mathsf{hyp}\ (Q\ Y'), u_2{:}\mathsf{hyp}\ (Q\ Y') & \vdash \mathsf{conc}\ (P\ (Y\ x))\quad \text{loop}
\end{array}$$

As in the previous example, we will end up in an infinite loop, however the subgoals now contain existential variables Y' and Y.

Although it is possible to design specialized propositional sequent calculus with loop detection [10], this often non-trivial and complicates the implementation of the proof search procedure. Moreover, proving the correctness of such a more refined propositional calculus, is non-trivial, because we need to reason explicitly about the structure of memoization. Finally, 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. Tabled logic programming provides generic memoization support for proof search and allows us to factor out common sub-proofs during proof search, thereby potentially obtaining smaller and more compact certificates. Since 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 and increases the expressive power and usefulness of the overall system. In the case of the propositional sequent calculus, we obtain a decision procedure for free.

3 Tabling in higher-order logic programming

Tabling methods eliminate redundant and infinite computation by memoizing subgoals and their answers in a table and re-using the results later. Our search is based on the multi-stage strategy by Tamaki and Sato [25], which differs only insignificantly from SLG resolution [4] for first-order logic programs without negation. Tabled search proceeds in stages and relies on a table to keep track of all the subgoals encountered, and answers which were derived for them. When trying to prove a goal G from the dynamic assumptions Γ, we first check if there exists a variant of $\Gamma \vdash G$ in the table. If yes, then we suspend the computation and backtrack. If no, we add $\Gamma \vdash G$ to the table, and proceed proving the goal using the dynamic assumptions in Γ and the program clauses. If we derive an answer for a goal $\Gamma \vdash G$, then this answer is added to the table. This first stage terminates, once all possible search paths have been explored, and the leafs in the search tree are either failure, success, or suspended nodes. In the next stage, we will re-consider the suspended nodes in the search tree, and try to grow the tree further by re-using answers of previous stages from the table. For a more

detailed description of the search we refer the reader to [19, 20]. Here we will discuss the basic design principles underlying tabled search, how to manage and access the table efficiently in the higher-order setting. These principles are largely independent of the actual strategy of how to reuse answers from the table. There are three main table access operations:

Call CheckInsert When we encounter a tabled subgoal, we need to check whether this subgoal is redundant. We check, if there exists a table entry $\Gamma' \vdash G'$ s.t. $\Gamma \vdash G$ (the current goal) is a variant (or instance) of the already existing entry $\Gamma' \vdash G'$.

Answer CheckInsert When an answer together with its proof witness is derived for a tabled subgoal, we need to check whether this answer has been already entered into the table answer list for this particular subgoal. If it has then the search fails, otherwise the answer together with its proof witness is added to the answer list, and may be re-used in later stages.

Answer Backtracking When a tabled subgoal is encountered, and answers for it are available from the table, we need to backtrack through all the answer.

A naive implementation can result in repeatedly rescanning terms and large table size thereby degrading performance considerably and rendering tabling impractical. This problem has been named *table access problem* in first-order logic programming [22]. In this section, we will describe design and implementation solution, which shares common structure and common operations in the higher-order setting using substitution factoring, linear higher-order patterns, higher-order substitution tree indexing, and compact proof witnesses.

3.1 Design of memo-table

The table records intermediate goals $\Gamma \vdash G$ together with answers and proof witnesses. As we have seen in the previous example, intermediate goals may refer to existential (or logic variables) which will be instantiated during proof search. In an implementation, existential variables are typically realized via references and destructive updates. This achieves that instantiations of existential variables are propagated immediately. On the other hand, we may need to undo these instantiations for existential variables upon backtracking. This is usually achieved by keeping a separate trail of existential variables and their corresponding instantiations. As a consequence, we must take special care in an implementation when memoizing and suspending the computation of intermediate goals. When suspending nodes, we copy the trail to re-instantiate the existential variables adapting ideas from [8]. Before storing intermediate goals in a memo-table, we must abstract over all the existential variables in a goal, to avoid pollution of the table. To illustrate, recall the previous subgoal:

$$x{:}\mathsf{i}, u{:}\mathsf{hyp}\ ((Q\ Y')\ \mathsf{and}\ (P\ x)) \vdash \mathsf{conc}\ (P\ (Y\ x))$$

To store this subgoal in a table, we abstract over the existential variables Y' and Y, to obtain the following table entry:

$$\begin{array}{ccccc} \Delta & ; & \Gamma & \vdash & G \\ y' : \mathsf{i}, y : \mathsf{i} \to \mathsf{i} & ; & x{:}\mathsf{i},\ u{:}\mathsf{hyp}\ ((Q\ y')\ \mathsf{and}\ (P\ x)) & \vdash & \mathsf{conc}\ (P\ (y\ x)) \end{array}$$

Δ refers to a context describing existential variables, Γ describes the context for the bound variables and dynamic assumptions and G describes the goal we are attempting to prove. To allow easy comparison of goals G with dynamic assumptions Γ modulo renaming of the variables in Δ and Γ, we represent terms internally using explicit substitutions [1] and de Bruijn indices.

Once this subgoal is solved and we inferred a possible instantiation for the existential variables in Δ, we will add the answer to the table. The answer is a substitution for the existential variables in Δ. In the previous example, the correct instantiation for Y is $\lambda x.x$, while the existential variable Y' is unconstrained. This leads to the following answer substitution:

$$\cdot \quad \vdash \quad (Y'/y',\ \lambda x.x/y) \quad : \quad y'{:}\mathsf{i}, y{:}\mathsf{i} \to \mathsf{i}$$

As we see in this example, not all instantiations for existential variables need to be ground. To avoid pollution of the answer substitution in the table, we again must abstract over the existential variables in the computed answer, which leads to the following abstracted answer substitution:

$$y'{:}\mathsf{i} \quad \vdash (y'/y',\ \lambda x.x/y) \quad : \quad y'{:}\mathsf{i}, y{:}\mathsf{i} \to \mathsf{i}$$

In general, the invariant about table entries and answer substitutions are:

Table entry	Answer substitution
$\Delta; \Gamma \vdash G$	$\Delta' \vdash \theta : \Delta$

The design supports naturally substitution factoring based on explicit substitutions[22]. With substitution factoring the access cost is proportional to the size of the answer substitution rather than the size of the answer itself. It guarantees that we only store the answer substitutions, and create a mechanism of returning answers to active subgoals that takes time linear in the size of the answer substitution θ rather than the size of the solved query $[\theta]G$. In other words, substitution factoring ensures that answer tables contain no information that also exists in their associated call table. Operationally, this means that the constant symbols in the subgoal need not be examined again during either answer checkInsert or answer backtracking. For this setup to work cleanly in the higher-order setting, it is crucial that we distinguish between existential variables in Δ and bound variables and assumptions in Γ.

To support selective memoization, we provide user keyword which allows the user to mark predicates to be tabled or not. If the predicate in G is not marked tabled, then nothing will change. We design the tabled search in such a way that it is completely separate from non-tabled search. The only overhead in non-tabled computation will be a check whether a given predicate is tabled.

3.2 Optimization: Linearization

A common optimization for first-order terms is linearization which enforces that every existential variable occurs only once. This means that any necessary consistency checks can be delayed and carried out in a post-processing step. In the

higher-order setting, we extend this linearization step to eliminate any computationally expensive checks involving bound variables and enforce that terms fall into the linear higher-order pattern fragment, where every existential variable occurs only once and must be applied to all the bound variables. Linear higher-order patterns refine the notion of higher-order patterns[11, 17] further and factor out any computationally expensive parts. As shown in [21], many terms encountered fall into this fragment and linear higher-order pattern unification performs well in practice. Consider again, the previous example:

$$\begin{array}{ccccc} \Delta & ; & \Gamma & \vdash & G \\ y' : \mathsf{i}, y : \mathsf{i} \to \mathsf{i} & ; & x{:}\mathsf{i},\ u{:}\mathsf{hyp}\ (Q\ x) & \vdash \mathsf{conc}\ (P\ (y\ x)) & \end{array}$$

Both occurrences of the existential variable y and y' are higher-order patterns, since they are applied to a distinct set of bound variables. However, the variable y' and y are not linear higher-order patterns, since neither is applied to all the bound variables which occur in Γ. During linearization, we will translate the goal into a linear higher-order pattern together with residual equations which factor out non-linear sub-parts. We abbreviate $y_1\ x\ u$[1] as $y_1[\mathsf{id}]$.

$$\begin{array}{ccccc} \Delta & ; & \Gamma & \vdash & G \\ y_1 : \mathsf{i} \to \mathsf{i} \to \mathsf{i}, & & & & \\ y_2 : \mathsf{i} \to \mathsf{i} \to \mathsf{i} & & & & \\ y' : \mathsf{i}, y : \mathsf{i} \to \mathsf{i} & ; & x{:}\mathsf{i},\ u{:}\mathsf{hyp}\ ((Q\ y_1[\mathsf{id}])\ \mathsf{and}\ (P\ x)) & \vdash \mathsf{conc}\ (P\ (y_2[\mathsf{id}])) & \end{array}$$

together with the residual equations R: $y_1[\mathsf{id}] \doteq y' \wedge y_2[\mathsf{id}] \doteq y\ x$
This motivates the final table design:

Table entry	Residual Equ.	Answer substitution
$\Delta; \Gamma \vdash G$	$\Delta; \Gamma \vdash R$	$\Delta' \vdash \theta : \Delta$

where G is a linear higher-order pattern, Γ denotes the bound variables and dynamic assumptions and Δ describes the existential variables occurring in G and Γ. This linearization step can be done together with abstraction and standardization over the existential variables in goal, hence only one pass through the term is required.

3.3 Optimization: Strengthening

We have seen previously that strengthening of the dynamic assumptions is necessary to prevent some loops. We previously concentrated on strengthening by removing duplicate assumptions from the dynamic context. However, in general we use in addition two other forms of strengthening based on a type-dependency analysis called subordination [26]. First, we eliminate dynamic assumption in Γ

[1] Intuitively, y_1 will be instantiated to a function $\lambda x.\lambda u.\lambda z.y_1$ and then β-reduced to$[u/u, x/x]y_1$. So y_1 denotes an open term, which may refer to the bound variables x and u. For a more formal treatment of open terms see [21]

which cannot possibly contribute to a proof of G. Second, we eliminate bound variable dependencies in existential variable.

Strengthening allows us to detect more loops during proof search and eliminate more redundant computation. Furthermore, it allows us to store some information more compactly.

4 Higher-order term indexing for tabling

To achieve an efficient and successful tabled logic programming interpreter, it is crucial to support efficient indexing of terms in the table to facilitate compact storage and rapid retrieval of table entries. Indexing techniques facilitate rapid retrieval of a set of candidates satisfying some property (e.g. unifiability, instance, variant etc.) from a large collection of terms. Although a wide range of indexing techniques exists in the first-order setting, indexing techniques for higher-order terms are almost not existent. The main problem in handling higher-order terms lies in the fact that most operations such as testing whether two terms are unifiable, computing the most specific generalization of two terms etc. are undecidable in the higher-order setting.

We propose substitution tree indexing for linear higher-order patterns. In [19, 20], we give a formal description for computing the most specific generalization of two linear higher-order patterns, for inserting terms in the index and for retrieving a set of terms from the index s.t. the query is an instance of the term in the index, and show correctness. Here we will concentrate on the adaptations to support tabling. The main algorithm of building a substitution tree follows the description in [23]. To illustrate higher-order substitution tree indexing let us consider the following set of linear higher-order patterns.

Goal	Residual Equation
$\mathsf{conc}\ (\mathsf{forall}\ \lambda z.\ ((P\ (f\ x)\ y_1[\mathsf{id}])\ \mathsf{and}\ \ Q\ \ z\ ))$	$y_1[\mathsf{id}] \doteq y[x/x]$
$\mathsf{conc}\ (\mathsf{forall}\ \lambda z.\ ((P\ \ z\ \ y_1[\mathsf{id}])\ \mathsf{and}\ \ Q\ \ y_2[\mathsf{id}]))$	$y_1[\mathsf{id}] \doteq y' \wedge\ y_2[\mathsf{id}] \doteq y[x/x]$
$\mathsf{conc}\ (\mathsf{forall}\ \lambda z.\ ((P\ \ z\ \ y_1[\mathsf{id}])\ \mathsf{and}\ \ Q\ \ y_2[\mathsf{id}]))$	$y_1[\mathsf{id}] \doteq y[\mathsf{id}] \wedge\ y_2[\mathsf{id}] \doteq y'$
$\mathsf{conc}\ (\mathsf{forall}\ \lambda z.\ ((P\ (f\ z)\ y_1[\mathsf{id}])\ \mathsf{and}\ Q\ (f\ y_2[\mathsf{id}])))$	$y_1[\mathsf{id}] \doteq y[\mathsf{id}] \wedge\ y_2[\mathsf{id}] \doteq y[x/x]$

For simplicity, we assume each of the goals has the same dynamic context $\Gamma = x{:}i, u{:}\mathsf{hyp}\ ((P\ y_0[\mathsf{id}]x)\ \mathsf{and}\ (Q\ x)$ and the context Δ describing the existential variables contains y_0, y_1, y_2, y, and y'. Below we show the substitution tree for the given set of linear higher-order patterns. Each table predicate will have its own substitution tree.

A higher-order substitution tree is a tree whose nodes are substitutions together with a context Δ_i which describes the existential variables occurring in the node. For example, the substitution in the top-most node contains the existential variable y_1, while the node with the substitution $f(i_3[\mathsf{id}])/i_1$ does not refer to any existential variable. It is crucial that we ensure that any internal variable i which is applied to all the variables in whose scope it occurs in. However, for any operation on the index, we must treat internal existential variables i differently than globally existential variables y. Internal existential variables i

will be instantiated at a later point as we traverse the tree. While existential variables defined in Δ_i are potentially subject to "global" instantiation, if we check whether the current goal is an instance of a table entry. The intention is that all the Δ_i along a given path together with the Δ_n at the leaf constitutes the full context of existential variables Δ. As there are no typing dependencies among the variables in Δ and all the variables in Δ are linear, they can be arbitrarily re-ordered. Distributing Δ along the nodes in the substitution tree, makes it easier to guarantee correctness in an implementation where variables are represented via de Bruijn indices.

$$\boxed{y_1{:}\ldots \mid \mathsf{forall}\ \lambda z.P(i_1[\mathsf{id}], y_1[\mathsf{id}])\ \mathsf{and}\ Q(i_2[\mathsf{id}])}$$

$$\boxed{\cdot \mid f(i_3[\mathsf{id}])/i_1} \qquad\qquad \boxed{y_2{:}\ldots \mid \begin{array}{r} z/i_1, \\ y_2[\mathsf{id}]/i_2 \end{array}}$$

$$\boxed{y_2{:}\ldots \mid \begin{array}{r} z/i_3 \\ f(y_2[\mathsf{id}])/i_2, \end{array}} \qquad \boxed{\cdot \mid \begin{array}{r} x/i_3, \\ z/i_2 \end{array}} \qquad \begin{array}{l} y_1[\mathsf{id}] \doteq y[\mathsf{id}] \\ y_2[\mathsf{id}] \doteq y' \end{array} \qquad \begin{array}{l} y_1[\mathsf{id}] \doteq y' \\ y_2[\mathsf{id}] \doteq y[\mathsf{id}] \end{array}$$

$$\begin{array}{l} y_1[\mathsf{id}] \doteq y\ x \\ y_2[\mathsf{id}] \doteq y\ x \end{array} \qquad y_1[\mathsf{id}] \doteq y[\mathsf{id}]$$

At the leafs, we will store linear residual equations, dynamic assumptions Γ, the existential variables Δ' occurring in the residual equations and in Γ, as well as a pointer to the answer list. Note we omitted the two latter parts in the figure above. By composing the substitutions along a path and collecting all the existential variables Δ_i along this path, we will obtain the table entry $\Delta; \Gamma \vdash G$ together with its residual equations. By composing the substitutions in the left-most branch, we obtain the term (4).

In the following development we will distinguish between internal existential variables i which are defined in the modal context Σ and "global" existential variables u and v which are defined in the modal context Δ. A higher-order substitution tree is an ordered tree and is defined as follows:

1. A tree is a leaf node with substitution ρ such that $\Delta_n \vdash \rho : \Sigma$.
2. A tree is a node with substitution ρ such that $(\Delta_j, \Sigma) \vdash \rho : \Sigma'$ and children nodes $N_1, \ldots, N_n$ where each child Node N_i has a substitution ρ_i such that $(\Delta_i, \Sigma_i) \vdash \rho_i : \Sigma$.

For every path from the top node ρ_0 where $(\Delta_0, \Sigma_1) \vdash \rho_0 : \Sigma_0$ to the leaf node ρ_n, we have $\Delta = \Delta_0 \cup \Delta_1 \cup \ldots \cup \Delta_n$ and $\Delta \vdash \rho_n \circ \rho_{n-1} \circ \ldots \circ \rho_0 : \Sigma_0$. In other words, there are no internal existential variables left after we compose all the substitutions ρ_n up to ρ_0. As there are no typing dependencies among the variables in Σ and all the variables are linear, they can be arbitrarily re-ordered.

At the leaf, we also store a list of answer substitutions θ, where we have $\Delta' \vdash \theta : \Delta$ and the dynamic context Γ. The design is motivated by the design of term

indexing in first-order theorem proving, where we would index and discriminate on the conclusion we want to prove rather than the assumptions. First, indexing on the goal, rather than on the context Δ and Γ, allows for more structure sharing, since often the goals are similar, but the context Γ and Δ may differ. In addition it allows direct access to the goal of the table entry and allows to implement a table for each type family.

5 Compact proof witnesses

Generating certificates as evidence of a proof is essential if we aim to use the tabled logic programming interpreter as part of a certifying code infrastructure. Moreover, it is helpful in guaranteeing correctness of the tabled search and debugging the logic programming interpreter. The naive solution to generate certificates when tabling intermediate sub-goals and their results, is to store the corresponding proof term together with the answer substitution in the table. However this may take up considerable space and results in high computational overhead, due to their large size[14]. Hence it is impractical to store the full proof term. In our implementation, we only store a footprint of the proof from which it is possible to recover the full proof term. Essentially we just keep track of the id of the applied clause thereby obtaining a string of numbers which corresponds to the actual proof. This more compact proof witness can be de-compressed and checked by building and re-running a deterministic higher-order logic programming engine. This idea to represent proof witnesses as a string of (binary) numbers is inspired by [13].

6 Experimental results

In this section, we discuss some experimental results with different examples. All experiments are done on a machine with the following specifications: 2.4GHz Intel Pentium Processor, 512 MB RAM. We are using SML of New Jersey 110.0.7 under Linux Red Hat 9. Times are measured in seconds.

6.1 Propositional Theorem Proving

We report on two experiments. The first one uses a straightforward implementation of the intuitionist sequent calculus, while in the second one we chain all invertible rules together and use focusing for the non-deterministic choices. In the straightforward implementation of the sequent calculus, we memoize every subgoal encountered. Although this is feasible and useful for testing smaller examples, the table size and especially the number of suspended goals may grow very large. In the focusing version of the propositional sequent calculus, we will only memoize subgoals once we come to the focusing phase, thereby controlling the table size. Moreover, we will employ strengthening. In our experiments, we consider all the propositional test-cases reported by J. Howe in [10], which he

used to evaluate his two specialized loop-detection mechanisms for intuitionist propositional logic. These examples provide a reality check on how powerful a generic proof search method with memoization can be. Due to space, we only can show some test results here.

Both implementations of the sequent calculus within Twelf will not be executable using a logic programming interpreter based on depth-first search, however it is possible to use the iterative deepening theorem prover which is part of the meta-theorem prover in Twelf [18]. Iterative deepening will stop after finding the first solution, hence we compare it to finding the first solution using tabled search. We also include the time it takes for tabled search to terminate, and conclude that no other solution exists.

Sequent Calculus Propositional Theorem Proving – Run time in sec

	ItDeep	Tab(1)	Tab(all)
$A \wedge (B \vee C) \vdash (A \wedge B) \vee (A \wedge C)$	1.48	0.11	0.15
$((A \vee B) \wedge (A \vee C)) \vdash (A \vee (B \wedge C))$	10.57	0.18	0.73
$A \vee C \wedge (B \supset C) \vdash (A \supset B) \supset C$	202.57	0.02	1.59

As we can see, iterative deepening takes considerable amount of time to prove de Morgan's laws, which are typical examples one would like to check. The table size for the tabled higher-order logic programming engine grows up to 560 table entries and over 3400 suspended nodes.

Unlike iterative deepening, where failure is not meaningful, we can use tabled logic programming to also disprove examples. The use of memoization yields a decision procedure for propositional logic for free.

Sequent Calculus Propositional Theorem proving – NonProvable

Name	Tab	Indices	SuspGoals
$(((A \supset B) \supset \mathsf{false}) \supset A \wedge (B \supset \mathsf{false}))$	0.01	15	34
$((A \wedge B \supset \mathsf{false}) \supset (A \supset \mathsf{false}) \vee (B \supset \mathsf{false}))$	0.03	31	80
$((A \supset B) \supset C) \vdash (A \vee C \wedge (B \supset C))$	0.01	19	42

We can improve upon a naive proof search procedure based on the sequent calculus, by chaining all the invertible rules such as andR, andL_2, andL_1, impR, orL together, and focusing on the choices we have to make in the impL and orR_1 and orR_2 rule. We will only memoize subgoals once we come to the focusing phase, thereby controlling the table size. The approach essentially builds on J. Howe work where he compared two different loop detection for intuitionist propositional logic and designed special propositional theorem provers for it. We will use the examples from his test-suite [10] to see how powerful a generic proof search engine with memoization can be.

Focusing Calculus (Propositional theorem proving) – run time in sec

Name	ItDeep	Tab(1)	Tab(all)
$(A \vee B) \wedge (D \vee E) \wedge (G \vee H) \supset (A \wedge D) \vee$ $(A \wedge G) \vee (D \wedge G) \vee (B \wedge E) \vee (B \wedge H) \vee (E \wedge H)$	0.230	0.05	0.05
$(((A \vee B \vee C) \wedge (D \vee E \vee F) \wedge (G \vee H \vee J) \wedge$ $(K \vee L \vee M)) \supset ((A \wedge D) \vee (A \wedge G) \vee (A \wedge K) \vee$ $(D \wedge G) \vee (D \wedge K) \vee (G \wedge K) \vee (B \wedge E) \vee (B \wedge H) \vee$ $(B \wedge L) \vee (E \wedge H) \vee (E \wedge L) \vee (H \wedge L) \vee (C \wedge F) \vee$ $(C \wedge J) \vee (C \wedge M) \vee (F \wedge J) \vee (F \wedge M) \vee (J \wedge M)))$	∞	4.12	4.23
$((((A \leftrightarrow B) \supset (A \wedge B \wedge C)) \wedge ((B \leftrightarrow C) \supset (A \wedge B \wedge C))$ $\wedge((C \leftrightarrow A) \supset (A \wedge B \wedge C))) \supset (A \wedge B \wedge C))$	∞	0.46	0.40
$(((D \supset (C \supset A)) \wedge A \wedge ((C \wedge A) \supset (C \wedge B)) \wedge ((A \vee$ $(C \supset A)) \supset (A \wedge D \wedge \neg(C \vee A)))) \supset (D \supset \neg(C \vee A)))$	∞	1.55	2.61
$((((C \supset D) \supset (A \wedge (B \supset B))) \wedge D \wedge (C \supset D) \wedge C \wedge$ $((B \supset A) \vee (C \wedge B))) \supset (((((C \supset C) \wedge (B \supset B)) \supset$ $(B \vee C)) \wedge D) \supset (((A \wedge D) \vee A) \supset ((C \wedge (B \supset B)) \wedge$ $(C \supset C) \wedge C) \supset \neg(\neg(B \vee C))))))$	22.12	0.38	0.40
$((\neg((D \supset C) \vee C \vee (A \wedge C \wedge (C \supset B))) \wedge$ $((D \wedge D) \vee ((D \wedge (C \supset A)) \supset \neg(C \vee A)) \vee$ $((B \wedge \neg B) \supset ((C \wedge C) \vee (D \supset B))))) \supset$ $(((D \wedge D) \supset C) \supset (((D \wedge (C \supset A)) \supset$ $\neg(C \vee A)) \vee ((B \wedge \neg B) \supset ((C \wedge C) \vee (D \supset B))))))$	5.14	0.35	17.37
$((((C \wedge (D \supset D)) \supset C) \wedge (C \supset D) \wedge (C \supset B)$ $\wedge C \wedge D \wedge C) \supset (((B \supset A) \wedge D) \supset ((A \vee C) \supset$ $((D \supset D) \supset ((B \supset A) \wedge (A \vee C \vee C)$ $\wedge((B \wedge C) \supset (B \wedge C)) \wedge (A \vee C))))))$	17.37	0.22	0.54
$(((C \supset C) \supset (B \wedge A)) \supset ((C \wedge D) \supset$ $((C \supset C) \supset ((C \wedge A \wedge D) \supset ((A \vee A) \supset$ $(\neg(C \supset (B \wedge A \wedge \neg A)) \vee ((A \vee A) \wedge B \wedge A \wedge C \wedge D \wedge$ $(((B \supset B) \supset D) \supset (A \wedge D)))))))))$	33.36	0.08	0.17
$(((((A \wedge A) \supset ((A \wedge A) \vee C)) \supset B) \wedge ((\neg(A \vee D) \supset$ $((C \supset B) \wedge (A \supset A))) \supset$ $(D \wedge ((C \supset B) \supset A))) \wedge \neg(D \supset D)) \supset (B \vee \neg A))$	0.1	1.15	53.50
$(\neg((((D \wedge A) \supset (A \supset C)) \vee (D \wedge C) \vee (D \supset B)) \wedge$ $\neg A \wedge \neg(((D \wedge A) \supset (A \supset C)) \vee (D \wedge C) \vee (D \supset B))))$	0.56	1.01	1.17

Focusing Calculus (Propositional Theorem Proving) – Disproving

Formula	tab
$(((A \vee B \vee C) \wedge (D \vee E \vee F)) \supset ((A \wedge B) \vee (B \wedge E) \vee (C \wedge F)))$	0.02
$(A \supset B) \supset ((A \supset B \supset C) \supset C) \supset (A \supset B \supset C)$	0.00
$((((\neg(\neg(\neg A \vee \neg B))) \supset (\neg A \vee \neg B)) \supset ((\neg(\neg(\neg A \vee \neg B))) \vee$ $\neg(\neg A \vee \neg B))) \supset (\neg(\neg(\neg A \vee \neg B)) \vee \neg(\neg A \vee \neg B)))$	11.99
$(((A \wedge (B \vee C)) \supset (C \vee (C \wedge D))) \supset ((\neg A) \vee ((A \vee B) \supset C)))$	0.01

Not surprisingly, iterative deepening is not powerful enough to prove most of the examples from Howe's test-suite. But tabled higher-order logic programming is able to prove or disprove 14 of the 15 propositional examples from Howe's test-suite in reasonable time frame. Most of the examples take 1 sec or below, and only one example took 4.12 sec. Only in one example the tabled logic programming interpreter started thrashing and was eating up too much memory. The table

size in these examples was up to 2600 table entries and up to 22000 suspended goals, which is the main limiting factor in these examples. These examples seem to indicate that to improve the tabled higher-order logic programming engine we need to be able to detect whether some suspended nodes are still productive or if they have been saturated and can be garbage-collected. In fact, this strategy is pursued in the XSB system. However, the fact that a generic higher-order proof search procedure with tabling can be used to prove or disprove so many non-trivial propositional examples, has been surprising to us. Maybe even more importantly, tabled higher-order logic programming can also disprove examples, thereby yielding a decision procedure for propositional logic for free.

6.2 Refinement type checking

In this example, we explore refinement type checking as described by Davies and Pfenning in [7]. This is an advanced type system for a small functional language MiniML where expressions may have more than one type and there may be many ways of inferring a type. The type system is executable with a depth-first logic programming interpreter, however the redundancy may severely hamper the performance. We will compare the performance between depth-first search and tabled search, and group the examples in three categories: 1) Finding the first solution 2) Discovering that a given program cannot be typed 3) Finding all possible solutions.

Refinement type checking – Typable examples (runtime in sec)

	Name	lp(1)	tab(1)	lp(all)	tab(all)
sub :	nat → zero → nat&zero → nat → zero& bit → bit → bit	0.10	0.31	3.91	0.36
sub :	((nat → pos → nat)&(pos → nat → nat)& (pos → pos → nat)&nat → nat → nat)	0.10	0.38	3.43	0.43
mult :	((pos → nat → nat)&(nat → nat → nat)& (nat → pos → nat)&(pos → pos → pos))	0.06	0.66	∞	0.84
square:	(pos → nat&nat → nat)	0.02	0.70	∞	1.06
square:	(pos → pos)	0.10	0.90	∞	0.88

– time out after 1h

Refinement type checking – Untypable examples

	Name	lp	tab
mult :	(nat → pos → nat)	805.97	0.35
plus :	((nat → nat → nat)&(nat → pos → pos)& (pos → nat → pos)&(pos → pos → zero))	8.14	0.20
mult :	((pos → nat → nat)&(nat → nat → nat)& (nat → pos → nat)&(pos → pos → zero))	∞	0.620
mult :	(nat → zero → pos)	289.11	0.30
square :	(pos → zero	∞	0.92
square :	(pos → zero&pos → nat	∞	0.72

– time out after 1h

As the results demonstrate, logic programming is superior, if we are only interested in finding the first solution, but is not able to disprove that a given program is in fact not well-typed. Similarly, finding all possible types for a given program is too unwieldy. The table contains up to 400 table entries and 300 suspended goals. The fact that depth-first-search is superior to tabled search is not surprising since managing the table imposes some computational overhead. Moreover, the tabled strategy delays the re-use of answers hence imposing a penalty. However, the tabled logic programming interpreter is able to solve all the examples within 1 sec. This attests to the strength and robustness of the system. Overall, our users attest to the fact that they rather take a reasonable performance hit, when finding the first solution, but obtain a more robust proof system, which is able to provide reasonably quick feedback.

6.3 Parsing

Parsing is a classic example to demonstrate the strength and usefulness of tabling. Often we want to mix right and left recursion to model right and left associativity in the grammar. This leads to specifications which are not executable using a depth-first search. Hence we compare iterative deepening with tabled search. In our example, we implemented a parser for parsing formulas into higher-order abstract syntax. As the results demonstrate, tabling is clearly superior to iterative deepening, and provides a practical way of experimenting with parsers and grammars. We only compare finding the first solution with tabling and finding the first solution with iterative deepening, and report on the time depending on the number of tokens parsed. Table size ranges up to 1500 table entries, and up to 1750 suspended goals.

Parsing: Provable – runtime

Name	#tokens	ItDeep	tab(1)
1	5	0.01	0.02
2	20	0.78	0.07
3	32	79	0.28
4	60	2820.02	0.94
6	118	∞	3.22
7	177	∞	7.75
8	236	∞	12.65

Time limit : 1h

Parsing: Not provable – runtime

Name	#tokens	Tab
1	19	0.01
2	31	0.27
3	58	0.50
4	117	2.24

6.4 Mini-ML Reduction Semantics

6.5 Mini-ML Reduction Semantics

Finally, we discuss the implementation of a small-step interpreter for a small functional language MiniML using higher-order logic programming. On top of a one-step relation, we define a reflexive transitive closure to chain multiple steps together. This evaluator is not directly executable using depth-first search, since

the transitivity rule will lead to an infinite loop. Hence we compare iterative deepening with tabled search. As the results demonstrate, tabled search is again superior to iterative deepening search, and we can execute and experiments with several interesting examples. Again the limiting factor of tabled search is the number of suspended goals which grows over 20,000. We compare the time it took the find the first solution. The examples are meant to represent simple test cases, one would write. Obviously, to create an efficiently evaluator the user would implement a big-step semantics. Nevertheless, it is important to test and experiment with a small-step semantics, since properties such as progress cannot be proven using the big-step semantics.

Small-step reductions (provable) – runtime

	Name	ItDeep	Tab(1)
5	`plus(0,1)`	0.25	0.01
9	`mult((0,0)`	0.25	0.02
7	`plus(1,1)`	∞	0.04
8	`plus(2,2)`	∞	0.11
10	`mult(1, 1)`	∞	1.61
13	`mult(1, 2)`	∞	6.48
18	`mult(1, 3)`	∞	13.68
19	`mult(1, 4)`	∞	29.45
15	`plus(2, minus(4,2))`	∞	75.72

Small-step reductions (unprovable) – runtime

	Name	Tab
6	`plus(0,2) = 1`	0.02
7	`plus(1, 1) = 3`	0.10
8	`plus(2,2) = 2`	0.27
10	`mult(1, 1) = 2`	6.27

7 Conclusion

In this paper, we described the design and implementation of a tabled higher-order logic programming interpreter within the Twelf system. The system including the test-suites is available at `http://www.cs.cmu.edu/~twelf` as part of the Twelf distribution. Crucial ingredients in the design are substitution factoring, linear higher-order patterns, higher-order substitution tree indexing, and compact proof witnesses. These techniques are the key ingredients to enabling tabling in higher-order logic programming or theorem proving systems. They should also be applicable to systems such as λProlog [12] or Isabelle [15].

The wide range of examples we have experimented with demonstrates that tabling is a significant step towards obtaining a more robust and more powerful proof search engine in the higher-order setting. Tabling leads to improved performance and more meaningful, quicker failure behavior. Improved failure behavior is important when prototyping and experimenting with formal systems

as we want to test our specifications with positive and negative test cases, to gain more confidence in it. Hence quick feedback is critical. This does not mean that tabling is a panacea for all the proof search problems, but rather the first step towards integrating and adapting some of the more sophisticated first-order theorem proving techniques to the higher-order setting.

Unlike most descriptions of tabling which rely on modifying the underlying WAM to enable tabling support, we have identified and implemented the essential tabling mechanisms independently of the WAM. Although we have tried to carefully design and implement tabling within the higher-order logic programming system Twelf, there is still quite a lot of room for improvements. The most severe limitation currently is due to the multi-stage strategy which re-uses answers in stages, and prevents the use of answers as soon as they are available. Different strategies have been developed in first-order tabled logic programming such as SCC scheduling (strongly connected components), which allows us to consume answers as soon as they are available and garbage collect unproductive suspended nodes [24]. In the future, we plan to adapt these techniques to the higher-order setting, and incorporate more first-order theorem proving techniques such as ordering constraints.

References

1. Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit substitutions. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 31–46. ACM, 1990.
2. Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *ACM Conference on Computer and Communications Security*, pages 52–62, 1999.
3. 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.
4. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, January 1996.
5. Karl Crary. Toward a foundational typed assembly language. In *30th ACM Symposiumn on Principles of Programming Languages (POPL)*, pages 198–212, New Orleans, Louisisana, January 2003. ACM-Press.
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 Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *International Symposium on Programming Language Implementation and Logic Programming (PLILP'98)*, volume 1490 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 1998.
7. 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.
8. Bart Demoen and Konstantinos Sagonas. CAT: The copying approach to tabling. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *International Symposium on Programming Language Implementation and Logic Programming*

(PLILP'98), Lecture Notes in Computer Science (LNCS), vol. 1490, pages 21–36, 1998.
9. 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.
10. Jacob M. Howe. Two loop detection mechanisms: a comparison. In *Proceedings of the 6^{th} Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, pages 188–200. Springer-Verlag, 1997. LNCAI 1227.
11. Dale Miller. Unification of simply typed lambda-terms as logic programming. In *Eighth International Logic Programming Conference*, pages 255–269, Paris, France, June 1991. MIT Press.
12. 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.
13. G. Necula and S. Rahul. Oracle-based checking of untrusted software. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 142–154, 2001.
14. George C. Necula and Peter Lee. Efficient representation and validation of logical proofs. In Vaughan Pratt, editor, *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS'98)*, pages 93–104, Indianapolis, Indiana, June 1998. IEEE Computer Society Press.
15. Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.
16. Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, June 1989. IEEE Computer Society Press.
17. 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.
18. 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 Lecture Notes in Artificial Intelligence (LNAI) 1632.
19. Brigitte Pientka. A proof-theoretic foundation for tabled higher-order logic programming. In P. Stuckey, editor, *18th International Conference on Logic Programming, Copenhagen, Denmark*, Lecture Notes in Computer Science (LNCS), 2401, pages 271 –286. Springer-Verlag, 2002.
20. Brigitte Pientka. *Tabled higher-order logic programming.* PhD thesis, Department of Computer Sciences, Carnegie Mellon University, December 2003. CMU-CS-03-185.
21. Brigitte Pientka and Frank Pfennning. Optimizing higher-order pattern unification. In F. Baader, editor, *19th International Conference on Automated Deduction, Miami, USA*, Lecture Notes in Artificial Intelligence (LNAI) 2741, pages 473–487. Springer-Verlag, July 2003.
22. 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.
23. I. V. Ramakrishnan, R. Sekar, and A. Voronkov. Term indexing. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, pages 1853–1962. Elsevier Science Publishers B.V., 2001.

24. 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.
25. 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.
26. Roberto Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, Available as Technical Report CMU-CS-99-167, Sep 1999.
27. David S. Warren. *Programming in tabled logic programming*. draft available from http://www.cs.sunysb.edu/~warren/xsbbook/book.html, 1999.