Efficient Retrieval of Subsumed Subgoals in Tabled Logic Programs

Flávio Cruz and Ricardo Rocha*

CRACS & INESC-Porto LA, Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
{flavioc,ricroc}@dcc.fc.up.pt

Abstract. Tabling based systems use call similarity to decide when a tabled subgoal should produce or consume answers. Most tabling engines do that by using variant checks. A more refined method, named call subsumption, considers that a subgoal A will consume answers from a subgoal B if A is subsumed by B, thus allowing greater answer reuse. Recently, we have developed an extension, called *Retroactive Call Subsumption*, that improves upon call subsumption by supporting bidirectional sharing of answers between subsumed/subsuming subgoals. In this paper, we present an algorithm to efficiently retrieve the set of currently evaluating subgoals that are subsumed by a more general subgoal.

1 Introduction

Tabled resolution methods solve some of the shortcomings of Prolog because they can considerably reduce the search space, avoid looping and have better termination properties than SLD resolution based methods [1]. Tabling works by memorizing generated answers and then by reusing them on *similar calls* that appear during the resolution process. In a nutshell, first calls to tabled subgoals are considered *generators* and are evaluated as usual, using SLD resolution, but their answers are stored in a global data space, called the *table space*. Similar calls are called *consumers* and are resolved by consuming the answers already stored for the corresponding generator, instead of re-evaluating them against the program clauses. There are two main approaches to determine if a subgoal A is similar to a subgoal B: *call variance* and *call subsumption*.

In call variance, A and B are similar if they can be identical through variable renaming. For example, $p(X, 1, Y)$ and $p(Y, 1, Z)$ are *variants* because both can be transformed into $p(VAR_0, 1, VAR_1)$. Tabling by call subsumption is based on the principle that if A is subsumed by B (i.e., if A is an instance or more specific than B) and S_A and S_B are the respective answer sets, therefore $S_A \subseteq S_B$. For example, subgoal $p(X, 1, 2)$ is subsumed by subgoal $p(Y, 1, Z)$ because there is a substitution $\{Y = X, Z = 2\}$ that makes $p(X, 1, 2)$ an instance of $p(Y, 1, Z)$. For some types of programs, call subsumption yields superior time performance, as it

* This work has been partially supported by the FCT research projects STAMPA (PTDC/EIA/67738/2006) and HORUS (PTDC/EIA-EIA/100897/2008).

allows greater reuse of answers, and better space usage, since the answer sets for the subsumed subgoals are not stored. Arguably, the most successful approach for subsumption-based tabling is the *TST (Time-Stamped Trie)* design [2].

Despite the advantages of using subsumption-based tabling, the degree of answer reuse depends on the call order of subgoals. If a more general subgoal is called before specific subgoals, answer reuse will happen, but if more specific subgoals are called before a more general subgoal, no reuse will occur. To solve this problem, we implemented an extension to the original TST design, called *Retroactive Call Subsumption (RCS)* [3], that supports subsumption-based tabling by allowing full sharing of answers between subsumptive subgoals, independently of the order they are called. RCS works by selectively pruning the evaluation of subsumed subgoals when a more general subgoal appears later on. In this paper, we describe the modifications made to the table space data structures and we discuss the new algorithm developed to efficiently retrieve the set of currently evaluating instances of a subgoal.

2 Retrieval of Subsumed Subgoals

2.1 Table Space Data Structures

Arguably, the most successful data structure for representing the table space is *tries* [4]. Tries are trees in which common prefixes are represented only once. Tries provide complete discrimination for terms and permit lookup and insertion to be done in a single pass. In a trie-based tabling system, each tabled predicate has a *table entry* that points to a *subgoal trie*. In the subgoal trie, each distinct trie path represents a tabled subgoal call and each leaf trie node points to a *subgoal frame*, a data structure containing information about the subgoal call.

In our new approach, each subgoal trie node was extended with a new field, named *in_eval*, which stores the number of subgoals, represented below the node, that are in evaluation. This field is used to, during the search for subsumed subgoals, prune the subgoal trie branches without evaluating subgoals, i.e., the ones with $in\_eval = 0$.

When a subgoal starts being evaluated, all subgoal trie nodes in its subgoal trie path get the *in_eval* field incremented. When a subgoal completes its evaluation, the path is decremented. Hence, for each subgoal leaf trie node, the *in_eval* field can be equal to either: 1, when the corresponding subgoal is in evaluation; or 0, when the subgoal is completed. For the root subgoal trie node, we know that it will always contain the total number of subgoals being currently evaluated.

When a chain of sibling nodes is organized in a linked list, it is easy to select the trie branches with evaluating subgoals by looking for the nodes with $in\_eval > 0$. But, when the sibling nodes are organized in an hash table, it can become very slow to inspect each node as the number of siblings increase. In order to solve this problem, we designed a new data structure, called *evaluation index*, in a similar manner to the *time stamp index* [2] of the TST design. An evaluation index is a double linked list that is built for each hash table and is

used to chain the subgoal trie nodes where the *in_eval* field is greater than 0. The evaluation index makes the operation of pruning trie branches much more efficient by providing direct access to trie nodes with evaluating subgoals. While advantageous, the operation of incrementing or decrementing a subgoal trie path is more costly, because these indexes must be maintained.

2.2 Matching Algorithm

The algorithm that finds the currently running subgoals that are subsumed by a more general subgoal S works by matching the subgoal arguments SA of S against the trie symbols in the subgoal trie T. By using the *in_eval* field as described previously, we can prune irrelevant branches as we descend the trie. When reaching a leaf node, we append the corresponding subgoal frame in a result list that is returned once the process finishes. If the matching process fails at some point or if a leaf node was reached, the algorithm backtracks to try alternative branches, in order to fully explore the subgoal trie T.

When traversing T, trie variables cannot be matched against ground terms of SA. Ground terms of SA can only be matched with ground terms of T. For example, if matching the trie subgoal $p(VAR_0, VAR_1)$ with the subgoal $p(2, X)$, we cannot match the constant 2 against the trie variable VAR_0, because $p(2, X)$ does not subsume $p(VAR_0, VAR_1)$.

When a variable of SA is matched against a ground term of T, subsequent occurrences of the same variable must also match the same term. As an example, consider the trie subgoal $p(2, 4)$ and the subgoal $p(X, X)$. The variable X is first matched against 2, but the second matching, against 4, must fail because X is already bound to 2.

Now consider the trie subgoal $p(VAR_0, VAR_1)$ and the subgoal $p(X, X)$. Variable X is first matched against VAR_0, but then we have a second match against a different trie variable, VAR_1. Again, the process must fail because $p(X, X)$ does not subsume $p(VAR_0, VAR_1)$. This last example evokes a new rule for variable matching. When a variable of SA is matched against a trie variable, subsequent occurrences of the same variable must always match the same trie variable. This is necessary, because the found subgoals must be *instances* of S. Therefore, this problem can be reduced to the task of finding all instances of S in trie T. To implement this algorithm, we use the following data structures:

- *WAM data structures*: we take advantage of the existent Prolog data structures based on WAM machinery: heap, trail, and associated registers. The heap is used to build structured terms, in which the subgoal arguments are bound. Whenever a new variable is bound, we trail it using the WAM trail;
- *term stack*: stores the remaining terms to be matched against the subgoal trie symbols;
- *term log stack*: stores already matched terms from the term stack and is used to restore the state of the term stack when backtracking;
- *variable enumerator vector*: used to mark the term variables that were matched against trie variables;

– *choice point stack*: stores choice point frames, where each frame contains information needed to restore the computation in order to search for alternative branches.

The procedure that traverses a subgoal trie and collects the set of subsumed subgoals of a given subgoal call can be summarized in the following steps:

1. setup WAM machinery and push subgoal arguments into the term stack.
2. fetch a term T from the term stack;
3. search for a trie node N where the *in_eval* field is not 0.
4. search for the next node with a valid *in_eval* field to be pushed on the choice point stack, if any;
5. match T against the trie symbol of N;
6. proceed into the child of N or, if steps 3 or 5 fail, backtrack by popping a frame from the choice point stack and use the alternative trie node;
7. once a leaf is reached, add the corresponding subgoal frame to the resulting subgoal frame list. If there are choice points available, backtrack to try them;
8. if no more choice point frames exist, return the found subsumed subgoals.

3 Conclusions

We presented a new algorithm for the efficient retrieval of subsumed subgoals in tabled logic programs. Our proposal takes advantage of the existent WAM machinery and data areas and extends the subgoal trie data structure with information about the evaluation status of the subgoals in a branch, which allows us to prune the search space considerably. We therefore argue that our approach can be easily ported to other tabling engines, as long they are based on WAM technology and use tries for the table space.

Initial experiments using the YapTab tabling engine with support for retroactive call subsumption, showed low overheads on programs that do not benefit from the new algorithm, when compared to traditional call subsumption, and very good results when applied to programs that take advantage of it [3].

References

1. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43**(1) (1996) 20–74
2. Johnson, E., Ramakrishnan, C.R., Ramakrishnan, I.V., Rao, P.: A Space Efficient Engine for Subsumption-Based Tabled Evaluation of Logic Programs. In: Fuji International Symposium on Functional and Logic Programming. Number 1722 in LNCS, Springer-Verlag (1999) 284–300
3. Cruz, F., Rocha, R.: Retroactive Subsumption-Based Tabled Evaluation of Logic Programs. In: European Conference on Logics in Artificial Intelligence. LNCS, Springer-Verlag (2010) To appear.
4. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming **38**(1) (1999) 31–54