On Just In Time Indexing of Dynamic Predicates in Prolog

Vítor Santos Costa

DCC-FCUP & CRACS-INESC Porto LA
Universidade do Porto, Portugal
`vsc@dcc.fc.up.pt`

Abstract. Prolog is the most well-known and widely used logic programming language. A large number of Prolog applications maintains information by *asserting* and *retracting* clauses from the database. Such dynamic predicates raise a number of issues for Prolog implementations, such as what are the semantics of a procedure where clauses can be retracted and asserted while the procedure is being executed. One advantage of Logical Update semantics is that it allows indexing. In this paper, we discuss how one can implement just-in-time indexing with Logical Update semantics. Our algorithm is based on two ideas: stable structure and fragmented index trees. By *stable structure* one means that we define a structure for the indexing tree that not change, even as we assert and as we retract clauses. Second, by *fragmented index tree* we mean that the indexing tree will be built in such a way that the updates will be local to each fragment. The algorithm was implemented and results indicate significant speedups and reduction of memory usage in test applications.

1 Introduction

Prolog is the most well-known and widely used logic programming language. Prolog is based on Horn Clauses, a subset of First Order Logic for which a refutation-complete procedure exists. Definite Horn clauses have a single positive literal, the head, and zero or more negative literals. Clauses are called facts, if they have no negative literals, or rules, otherwise. Prolog programs rely on a static data-base of facts and procedures, but a large number of applications maintains information by *asserting* and *retracting* clauses from the database. In modern Prolog systems, predicates where one can assert and retract clauses are named *dynamic*.

A major issue for Prolog implementers is what are the *semantics* of a procedure where clauses can be retracted and asserted while the procedure is being executed. Originally, Prolog systems implemented what is called *immediate* updates, where a new clause or a retracted clause was immediately visible to a new execution. Unfortunately, these semantics can be hard to capture. Indexing also becomes impossible, as we may need to backtrack to a newly added clause.

Lindholm and O'Keefe [5] proposed what is nowadays called the *Logical Update semantics.* In these semantics, a call to a dynamic predicate operates on a snapshot of the clauses defining a predicate at the time of the call. In other words, adding or removing clauses will not change the clauses seen by a call to a predicate. The approach allows indexing and is intuitive. On the other hand, one problem is that deleted clauses need to be stored until all calls that use them terminate; moreover, the current goal should not be allowed to backtrack into "future" clauses. Lindholm and O'Keefe use time-stamps to control access to clauses, and garbage collection to remove dead clauses.

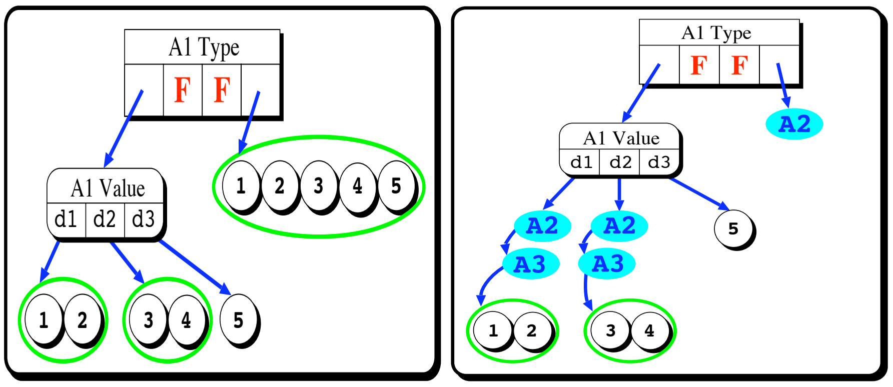
Logical Update semantics have become very popular. They are enforced by the ISO Prolog standard and they are implemented in most Prolog systems [2] (although not all [6]). As discussed above, one advantage of LU semantics is that it allows indexing (in contrast with immediate semantics). In this paper, we discuss how one can implement just-in-time indexing with logic update (LU) semantics. Just-in-time indexing [8] was motivated by the observation that the performance of Prolog programs strongly depends on the ability of the Prolog engine to find sets of clauses to match against sub-goals. Unfortunately, most Prolog systems have very limited indexing, usually by default just on the first argument. Extensions, such as multi-argument indexing [12], require user intervention and are designed to optimize a single mode of usage. *Just in Time Indexing (JITI)*, can address larger databases, allowing for different modes of usage. JITI relies on two key ideas. First, indexing should be performed *late*, when we know how the actual queries are instantiated. Second, different queries may have different modes. Thus, the indexing code should be able to change and *grow* to support different moded queries.

Dynamic predicates present a challenge to JITI: the indexing code can grow both because we have new modes of usage, and because we have new clauses. Moreover, the code can now *contract* as we retract clauses. Next we contribute over prior work [8] by extending the JITI algorithm to index dynamic predicates. The new algorithm is based on two ideas: stable structure and fragmented index trees. By *stable structure* one means that we define a structure for the indexing tree that not change, even as we assert and as we retract clauses. Second, by *fragmented index tree* we mean that the indexing tree will be built in such a way that the updates will be local to each fragment. The algorithm was implemented on the YAP Prolog system and results indicate significant speedups and reduction of memory usage in test applications.

The paper is organized as follows. First, we briefly review the JITI through an example. Next, we discuss the updating algorithm. We present performance results next. Last, we present some conclusions and suggest further work.

2 The JITI by Example

The JITI extends David H. D. Warren's WAM design. Figure 1(a) shows the WAM code for a small database fragment from a well-known machine learning



(a) (b)

Fig. 1. WAM and JITI Code for `has_property/3`. The “Type” nodes have four children: constant, compound term, list or unbound term. Sets of clauses are grouped as ovals.

dataset, *Carcinogenesis*, which contains information on the carcinogenic properties of several chemical compounds in mice [10]:

```
has_property(d1, salmonella, p).
has_property(d1, salmonella_n, p).
has_property(d2, salmonella, p).
has_property(d2, cytogen_ca, n).
has_property(d3, cytogen_ca, p).
```

Figure 1(a) shows WAM indexing code as a tree, with *switch* nodes, and *clause chain* nodes. The WAM has two different switches: *switch_on_type* selects according to the first argument, A_1, being unbound, constant, pair or structure; *switch_on_constant* selects according to a value. The WAM also separates *clause chain* nodes with a single clause, that are implemented as a jump, and sets of clauses, that are implemented as `try`, `retry`, `trust` sequences of instructions.

In the original WAM one only considers A_1. This is not always optimal, as shown in the following queries:

```
?- has_property(d1, _, _).
?- has_property(d1, salmonella, _).
?- has_property(_, salmonella_, _).
?- has_property(_, cytogen_ca, p).
?- has_property(_, _, n).
```

The WAM would only generate ideal code for the first example. In all the other cases it would backtrack through every clause in the procedure, even though the queries are deterministic.

Unfortunately, generating indexing code for every combination of arguments can be quite expensive: with arity 3 and no sub-arguments, we have $2^3 = 8$ different input-output combination. The JITI approaches this task by assuming that all calls will have the same mode of usage: it generates multi-argument indices based on the indexing patterns for the *first* call of a predicate.

Figure 1(b) shows JITI code for our example procedure, if the first query was:

```
?- a(d1,Prop,Type).
```

The JITI algorithm generates the code at the first call. The code is based on the WAM code, but it differs as follows:

– If A_1 = `d1` or A_1 = `d2` the JITI cannot commit to a single clause. In order to achieve determinacy, it thus adds extra code to test whether A_2 is instantiated. If the test succeeds, it tries generating a new index for the remaining clauses on A_2. This is not the case, so the JITI again adds code to test whether A_3 is instantiated. In the example, A_3 is unbound, so the JITI has no choice but to generate a `try-trust` chain.
– If A_1 = `d3` there is no need to perform further indexing: the goal is already determinate so the JITI transfers control to clause 5.
– The WAM generates a sequence of `try_me`, `retry_me`, `trust_me` instructions for the case A_1 is unbound. In the purest version of the JITI this is not necessary: the JITI just leaves code to check for the next argument, A_2.

We call the nodes that wait until an argument instantiates to generate code for the argument *wait* nodes, because the JITI is waiting on a chance to increase the trees. The usefulness of wait nodes becomes clear when consider the next example goal:

```
?- a(d1,salmonella,Type).
```

In this case, we have A_2 bound. Going back to Figure 1(b), we execute the `switch_on_type` instruction, next the `switch_on_cons`, and enter the wait node for A_2. At this point, the indexing code expands `d1`'s sub-tree using the same algorithm we used to build the indexing code for A_1. It first generates a `switch_on_type`, and then a `switch_on_cons`. The new tree replaces the A_2 wait node, resulting in the tree shown in Figure 2(a). Notice that we only expand the case for `d1`: we could be aggressive and also expand for `d2`, but our reference implementation tries to generate code as lazily as possible.

Last, imagine we run a call where only the second argument is bound:

```
?- a(Compound,salmonella,Type).
```

again we will execute the top `switch_on_type`, but now we will proceed to the rightmost wait node. The JITI builds a new indexing tree for A_2, shown in Figure 2(b).

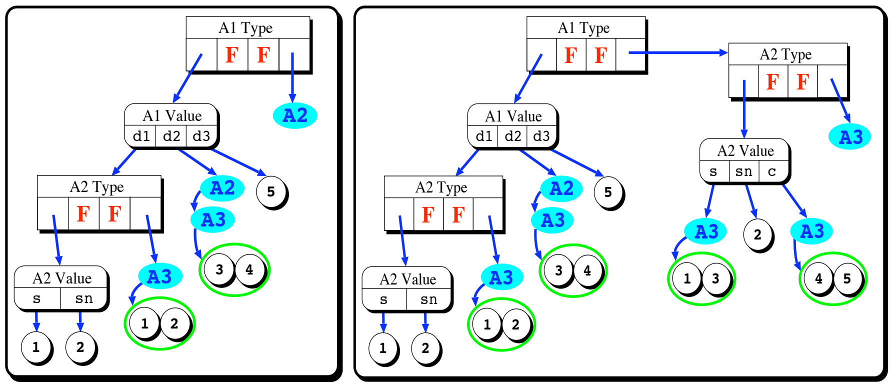


Fig. 2. Expanding JITI Code for `has_property/3`: A_1, A_2 bound and A_2 bound

3 Dynamic Procedures

We have explained the key ideas in the JITI. Before we proceed, it is important to understand how the YAP Prolog system implements dynamic predicates. Dynamic procedures are sets of dynamic clauses. Each dynamic clause is always allocated *independently* (in contrast to static clauses [7]). The main fields in the clause's header are:

– `ClFlags`: every clause has a set of flags; important flags indicate whether the clause has been deleted, and whether it is in use.
– `ClRefCount`: how many external pointers to the clause; most of these pointers will be from indexing code.
– `ClSource`: a data-structure containing the original source of the clause.
– `ClPrev, ClNext`: pointers to a doubly linked list.

The `assert/1` procedure stores a clause with three components: the header, the compile term, and a term containing the source-code.

The `retract/1` and `erase/1` procedures remove a clause from a list immediately, but do not try to recover space. Space is only recovered when `ClFlags` does not have `in-use` set, and if the `ClRefCount` is 0. Actually recovering space is the performed either at backtracking or when recovering space in the indexing code.

Our system implements a few optimizations of interest to dynamic code:

– Facts do not have source code: source code is actually recovered by executing the code;
– The Prolog Internal Data-Base is implemented as dynamic clauses with a single instruction, that unifies the second argument with a copy of the data in the `ClSource` field.

4 Indexing Dynamic Procedures

We have so far presented the JITI assuming the code is static. Dynamic procedures introduce several complications. The key ideas in our implementation are as follows: **(a)** try to keep the code as simple as possible; **(b)** try to make as little changes to the current tree as possible; **(c)** try to make changes local at each node. We discuss these principles next.

Simplicity Our discussion so far assumes that indexing code is always a simple tree, rooted at a `switch_on_type` node, with switch on value nodes below, and where leaves are wait nodes, jumps to clauses, or sequences of clauses. We shall always follow this structure for dynamic procedures, even though this is not always true in the WAM and in the JITI for static procedures. For example indexing code on A_1 for

```
has_property(_, salmonella, p).
has_property(d1, salmonella_n, p).
has_property(d2, salmonella, p).
has_property(d2, cytogen_ca,n).
has_property(_, cytogen_ca, p).
```

first *tries* the clause, then *retries* and enters a switch, and last *trusts* the fifth clause. Updating such sequences is difficult, and not typical of the code most often found with dynamic procedures.

Structure Preservation For simple trees, asserting a clause will not affect the *structure* of the tree: we just expand switch on values nodes, and add new leaves, but we do not need to build any *new* type and value switches tables. Retracting a clause is more complex: if a clause is the only clause for a sub-tree, we can (and should) recover space for the whole sub-tree.

Notice that the term *last* clause in a sub-tree must be understood in the context of LU semantics: we can only remove a clause from a `try–trust` when we are sure the code will not backtrack there.

Locality Last, given that the tree structure is preserved, adding a clause can be seen as a set of independent updates to non-structure nodes: we just need to visit the tree and apply an operation at each node. Retracting follows similar principles, but with the caveat that we must remove empty sub-trees.

4.1 Asserting Clauses

We can now present a clause insertion algorithm, $Insert(p, C, B)$, where p is the predicate, C the new clause, and B a boolean saying whether we want to insert as the *first* or as the *last* clause.

Our algorithm performs a pre-order walk of the indexing tree. Throughout, it maintains two stacks: the *label stack* includes pointers to all the branches we

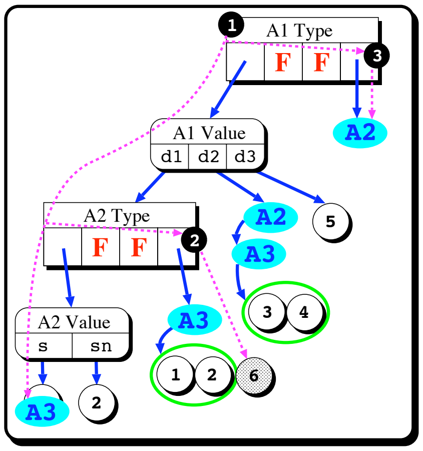


Fig. 3. Assert on `has_property/3`

have yet to visit; the *block stack* contains the node's parents. Both stacks are initially empty. The algorithm proceeds as follows:

1. If the current node is a `switch_on_type` node on argument A_i, then:
   (a) check what constraints the new clause imposes on A_i.
   (b) If the clause imposes no constraints or if A_i can match different types (e.g., $number(A_i)$), remove the sub-tree rooted at this node, replacing it by a `wait` node. One could add a `try` before/or a `trust` after the `switch_on_type` but doing so would break simplicity, as discussed above.
   (c) Otherwise, the clause must be added to two of the four cases in the `switch_on_type`: the case that matches the type, and the unbound case. The label stack allows us to process this case: we push the unbound label to the *label stack*, and jump to the bound case.
2. If the current node is a `switch` on value node, say `switch_on_constant`, then there must be a `switch_on_type` above, and that node ensured there is a constraint of the form $A_i = V$ in the clause:
   (a) if V is not in the table, one must expand the table. Our system follows the WAM and has a number of different cases: if the table is small, it use sequential search; otherwise, it uses a hash table. In either the case, if there is room in the table, one simply can insert the new entry. If there is no room, a new table is allocated. The system guarantees that value tables are only pointed from the current instruction, so the old table can be removed immediately. The algorithm than proceeds by popping the next instruction from the label stack.
   (b) if V is in the table, and matches a single clause (leaf), replace V's code by a wait node. Again, pop the next instruction from the label stack.
   (c) Otherwise, push the current block on the block stack and follow the table entry.

3. If the current node is a `try-retry-trust` chain (TTT): add a pointer to the clause to the chain. Our algorithm represents TTT nodes as lists of clauses [1], so we simply need to update the predicate's time-stamp and add the instruction to the list.
4. If the current node is a childless `wait` node: pop next instruction from label stack if childless. Otherwise, proceed to the node's child.

The algorithm terminates when the label stack empties.

Figure 4.1 shows an example of how the algorithm works, given the tree in Figure 2(a).

```
?- assert(atm(d1, salmonella, n)).
```

The algorithm starts by visiting the top `switch_on_type` node. The algorithm detects the constraint A_1 = `d1`, hence it takes the constant label (shown as path 1) and pushes the unbound pointer to the label stack. Next, it visits the value node for A_1. The `d1` case points to a tree, so there is no need to update the node. Next, it visits the `switch_on_type` for A_2. It detects the constraint A_2 = `salmonella`, hence the code will do as for A_1. It takes the leftmost branch, and pushes the rightmost pointer to the stack. The next `switch_on_constant` has two cases, `salmonella` and `salmonella_n`. The `salmonella` case pointed to a single clause, which must be replaced by a pointer to a `wait` node.

The next step is to pop the label stack and enter path 2. The algorithm first meets a wait node $\mathcal{T}_2$ with a child. It follows the child and meets the TTT chain node. At this point, it simply adds the new clause as a last `trust` in the chain, and replaces the previous `trust` clause 6 and the `trust 2` by a `retry 2`. We pop again and enter path 3. We only find a `wait`, pop again and exit.

4.2 Retracting Clauses

The first difference between the retracting and asserting, is that when retracting *we know that the clause must be consistent with the indexing tree.* Again, we start by emptying the two stacks, and enter at a `switch_on_type` node. The algorithm then does a pre-order walk in a fashion similar to the previous algorithm:

1. If the number of clauses for the predicate was 2, remove the whole tree.
2. If the current node is a `switch_on_type` node on some argument A_i, we must have a constraint of the form $A_i = V$. Push the unbound label to the label stack, and jump to the label matching V's type.
3. If the current node switches on value, then either:
   (a) V matches a single clause, replace the entry with `fail`.
   (b) Otherwise, jump to the label in the table, pushing the previous block on the block stack.
4. If the current node is a TTT node we have a number of possibilities:
   (a) 2 nodes in the chain: mark the TTT chain as deleted and replace the entry above by a pointer to the last clause.
   (b) else, if the TTT chain is not in use: remove the clause;
   (c) else, mark the TTT chain as *dirty*.

As usual, the algorithm terminates when the label stack empties.

	Time (msec)	Dynamic Code (in KB)					
		Clauses	*Indices*	*Switches*	*TTT*	*Wait*	*Tables*
1st **Arg**	211	998	126	7	85	18	14
Full	181	998	165	10	98	44	20

Table 1. Running Time and Space Usage for Dynamic Predicates in the Kiraz/Grimley-Evans benchmark at the end of forward execution. Sizes refer to total size spent in dynamic clauses and in indices. Indexing code is divided into code for `switch_and_type` and `switch_on_value` instructions, sequences of `try-retry-trust` instructions, `wait` nodes, and indexing tables.

Purging Dynamic Indices Dynamic nodes are purged after retracting clauses or after backtracking. Reclaiming a node is only performed after detaching all children; decrease reference to other clauses, and decrease the parent's reference counter.

5 Performance

The performance of JITI has been discussed in [8]. Previous results [3] show that for an ILP system the benefits of indexing are partly derived from indexing dynamic procedures. Next, we present in more detail some time and performance results on two real applications typical of dynamic updates, one from van Noord's FSA utilities [11], and the other from the ILP system Aleph [9].

The experiments were performed using on a Max Powerbook Pro with a 2.5GHz Intel Core Pro Duo and 4GB of memory, running OSX Leopard in 32 bit mode. The experiments consisted of running applications with the JITI totally enabled and only enabled for the main functor of the first argument. They therefore always apply the JITI.

Kiraz and Grimley-Evans The FSA toolkit includes a number of algorithms for finite-state automata. One of these algorithms is based on Kiraz and Grimley-Evans' algorithm for automata minimization [4]. The algorithm maintains a state table as dynamic procedures, and operates by manipulating this table, therefore it strongly depends on indexing.

Table 1 shows an advantage of using full versus first argument indexing in this case: there is a 10% speedup, at the cost of a 20% increase in index space. On the other hand, index space is still much smaller than clause compiled space.

A more detailed analysis finds that the difference essentially comes from two predicates, `symbol_state_/2` and `class_state_/2`. If one uses multiple argument, indexing execution is deterministic, and no choice-points are created. If one just uses first argument indexing, execution is non-deterministic. Execution thus becomes slower for two reasons: the system has to create choice-points, and it has to maintain a TTT chain for `symbol_state_/2` and `class_state_/2`. At the end of execution, the application erases all clauses for `symbol_state_`.

Experiment		Time (msec)	Dynamic Code (in KB)					
			Clauses	*Indices*	*Switches*	*TTT*	*Wait*	*Tables*
`sat(1)`	1st **Arg**	23	1066	25	4	4	9	7
	Full	21	1066	36	11	2	17	4
`reduce`	1st **Arg**	4385	12170	238	5	4	139	7
	Full	4100	12107	1495	12	91	148	23
`sat(2)`	1st **Arg**	34	12182	158	4	6	139	7
	Full	28	12182	291	16	94	153	27
`reduce`	1st **Arg**	4710	11471	162	5	6	143	7
	Full	4270	11471	378	17	155	157	49
`sat(3)`	1st **Arg**	17	11437	154	4	4	137	7
	Full	14	11437	369	13	153	147	46
`reduce`	1st **Arg**	4980	10713	284	5	4	267	7
	Full	4090	10713	595	13	242	277	61

Table 2. Running Time and Space Usage for a possible Aleph execution where we first saturate example 1, reduce, saturate example 2, reduce, saturate example 3, and reduce.

Using multiple arguments results in deeper and larger trees. The results show that *Switches*, the space spent on `switch_on_type` and the space spent on the non-table code in the `switch_on_cons` instructions, grows to 10KB from 7KB. The last interesting data concerns the sizes of the tables used to store hash tables and other chains. The results show *Table* size of 20KB for multiple argument, whereas when using the first argument usage is negligible. The `wait` node space corresponds to the space spent to support quick expansion of indexing trees. This space will grow as we generate deeper trees.

In a second experiment we use the *Aleph* Inductive Logic Programming System running the `Carcinogenesis` benchmark [10]. We run saturation and reduction three times, for three different examples. Because the Carcinogenesis database strongly benefits from multiple indexing, the 1st argument experiment only disables indexing on the dynamic predicates.

Table 2 shows performance for a possible Aleph run where one saturates the first example, reduces, the second example, reduce. The results are quite similar to the previous dataset. There is a speedup in using multiple argument indexing, at the cost of using more memory, but the total amount of memory is still quite low compared to the compiled code. A detailed analysis shows again two predicates dominating execution in the multiple-argument case: `$aleph_search_expansion/4` and `$aleph_search_node/8`. The latter predicate performs well with first argument indexing, but the second one creates very large TTT chains.

6 Discussion

We discuss the implementation of dynamic updates for just-in-time indexing. Although the key ideas have been presented previously, our contribution provides a detailed explanation of the actual algorithms, given wide experience in using the system.

Experience has forced a number of changes over the initial implementation. Arguably, the main change was that the original implementation maintained TTT chains as a single block. Extra space was reserved at the beginning and end in case clauses would be asserted. Instead of using time-stamps, blocks were copied. Unfortunately, experience showed that copying was just too expensive for larger applications.

In general, TTT chains have shown to be the source for most of the complexity of in the system. Note that the WAM always has a TTT chain, where each instruction is stored just before a clause [13]. By default, our system has no TTT chains, both on static and dynamic procedures. Clauses are linked by an single chain, for static procedure, and by a doubly linked lists, for dynamic procedures. One advantage of using a default chain is that it is straightforward to enumerate every clause. In contrast, enumerating clauses requires generating a new index, which is expensive for predicates with large number of clauses. In the worst case, calling a built-in such as `listing/0` might overflow the system. The system chose not to support a default TTT chain in order to save space in large databases [7].

Indexing dynamic clauses can be seen as a "light" version of the static indexing algorithm. This "simplicity" principle is based on experience: updating in the original WAM algorithm was complex because it is not always very clear where a TTT chain ends, and thus, what is a real `trust` instruction.

There is relatively little work on indexing dynamic procedures. Some inspiration to our work comes from the way indexing is implemented in SICStus Prolog [1]: SICStus uses very dynamic data-structures to support logical updates. One interesting alternative to our approach would be to support a single mode: XSB supports indexing dynamic data through tries [6]. Tries are a compact representation, and are clearly much more space efficient than our approach. On the other hand, they cannot support logical updates and multiple modes.

7 Conclusions and Future Work

We present in detail an indexing algorithm for Prolog that can support Just-In Time Compilation and that can support Logical Update Semantics. The algorithm was motivated by experience with user programs, and has been successfully applied in practice. The major advantage of this approach is that it supports the main benefits of the JITI in applications such as updatable data-bases of ground terms. The main cost in our experience is that it can be relatively expensive in terms of memory usage: namely, allocating independently each instruction in a TTT chain is quite costly space-wise.

Our implementation includes a number of design decisions that were based on particular applications. We expect that new applications will require further improvements. Ultimately, though, the question is whether one should invest on alternate data-structures that can provide the functionality of dynamic procedures in a more disciplined and efficient fashion.

Acknowledgments

This work has been partially supported by projects JEDI (PTDC/EIA/66924/2006) and STAMPA (PTDC/EIA/67738/2006) and funds granted to CRACS through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSC.

References

1. M. Carlsson. Freeze, Indexing, and Other Implementation Issues in the Wam. In *ICLP87*, pages 40–58, May 1987.
2. P. Deransart, A. Ed-Dbali, L. Cervoni, and A. A. Ed-Ball. *Prolog, The Standard: Reference Manual*. Springer Verlag, 1996.
3. N. A. Fonseca, V. Santos Costa, R. Rocha, R. Camacho, and F. M. A. Silva. Improving the efficiency of inductive logic programming systems. *Softw., Pract. Exper.*, 39(2):189–219, 2009.
4. G. A. Kiraz and E. Grimley-Evans. Multi-Tape Automata for Speech and Language Systems: A Prolog Implementation. In *Automata Implementation, LNCS 1436*. Springer-Verlag, 1998.
5. T. G. Lindholm and R. A. O'Keefe. Efficient implementation of a defensible semantics for dynamic Prolog code. In J.-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, MIT Press Series in Logic Programming, pages 21–39. University of Melbourne, "MIT Press", May 1987.
6. K. F. Sagonas, T. Swift, D. S. Warren, J. Freire, and P. Rao. The XSB programmer's manual. Technical report, State University of New York at Stony Brook, 1997. Available at http://xsb.sourceforge.net/.
7. V. Santos Costa. Prolog performance on larger datasets. In M. Hanus, editor, *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007, Nice, France, January 14-15, 2007.*, volume 4354 of *Lecture Notes in Computer Science*, pages 185–199. Springer, 2007.
8. V. Santos Costa, K. Sagonas, and R. Lopes. Demand-driven indexing of prolog clauses. In V. Dahl and I. Niemelä, editors, *Proceedings of the 23rd International Conference on Logic Programming*, volume 4670 of *Lecture Notes in Computer Science*, pages 305–409. Springer, 2007.
9. A. Srinivasan. *The Aleph Manual*, 2001.
10. A. Srinivasan, R. King, S. Muggleton, and M. Sternberg. Carcinogenesis predictions using ilp. In S. Džeroski and N. Lavrač, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 273–287. Springer-Verlag, 1997.
11. G. van Noord. FSA Utilities: A Toolbox to Manipulate Finite-State Automata. In *WIA: International Workshop on Implementing Automata, LNCS*. Springer-Verlag, 1997.
12. P. Van Roy, B. Demoen, and Y. D. Willems. Improving the execution speed of compiled Prolog with modes, clause selection and determinism. In *TAPSOFT'87*, pages 111–125. Springer Verlag, 1987.
13. D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.