*TLP* **12** (1–2): 5–34, 2012. © Cambridge University Press 2011
doi:10.1017/S1471068411000512 First published online 18 October 2011


# *The YAP Prolog system*

VÍTOR SANTOS COSTA and RICARDO ROCHA
*DCC & CRACS INESC-Porto LA, Faculty of Sciences, University of Porto*
*R. do Campo Alegre 1021/1055, 4169-007 Porto, Portugal*
(*e-mail:* {vsc,ricroc}@dcc.fc.up.pt)

LUÍS DAMAS
*LIACC, Faculty of Sciences, University of Porto,*
*R. do Campo Alegre 1021/1055, 4169-007 Porto, Portugal*
(*e-mail:* luis@ncc.up.pt)

*submitted 10 October 2009; revised 5 March 2010; accepted 6 February 2011*

**Abstract**

Yet Another Prolog (YAP) is a Prolog system originally developed in the mid-eighties and that has been under almost constant development since then. This paper presents the general structure and design of the YAP system, focusing on three important contributions to the Logic Programming community. First, it describes the main techniques used in YAP to achieve an efficient Prolog engine. Second, most Logic Programming systems have a rather limited indexing algorithm. YAP contributes to this area by providing a dynamic indexing mechanism, or just-in-time indexer. Third, a important contribution of the YAP system has been the integration of both or-parallelism and tabling in a single Logic Programming system.

*KEYWORDS*: Prolog, logic programming system

## 1 Introduction

Prolog is a widely used Logic Programming language. Applications include the semantic web (Devitt *et al.* 2005), natural language analysis (Nugues 2006), bioinformatics (Mungall 2009), machine learning (Page and Srinivasan 2003), and program analysis (Benton and Fischer 2007), just to mention a few. In this paper, we discuss the design of the *Yet Another Prolog (YAP)* system and discuss how this system tries to address the challenges facing modern Prolog implementations. First, we present the general structure and organization of the system, and then we focus on three contributions of the system to the Logic Programming community: *engine design*, the *just in-time indexer* (JITI), and *parallel tabling*. Regarding the first contribution, one major concern in YAP has always been to maintain an efficient interpreted Prolog engine. The first implementation of the YAP engine achieved good performance by using an emulator coded in assembly language. Unfortunately, supporting a large base of assembly code raised a number of difficult portability and maintenance issues. Therefore, more recent versions of YAP use an emulator written in `C`. A significant contribution of our work was to propose a number of techniques for

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

Prolog emulation and show that these techniques can lead to significant increases in the performance (Santos Costa 1999). Although our initial concern was execution speed, memory usage is also a significant issue in several Prolog applications, namely if the applications manipulate large databases. YAP implements a number of techniques to reduce the total memory usage in this case (Santos Costa 2007).

Ideally, Logic Programming should be about specifying the logic of the program, and then provide control. In practice, Logic Programming systems can often be very vulnerable to seemingly irrelevant details such as argument order. Especially for larger databases, swapping order of arguments may result in order of magnitude speed improvements for some programs. As such databases become more common, these problems become more important. YAP contributes to this area by providing a dynamic indexing mechanism, or JITI (Santos Costa *et al.* 2007). The JITI alleviates the questions of argument order, as it can make Prolog competitive in applications that would otherwise require a database manager (Davis *et al.* 2005).

A third contribution of the YAP system has been the integration of *both* or-parallelism and tabling in a single Logic Programming system. Inspired by previous research on the Muse system (Ali and Karlsson 1990) and the XSB engine (Sagonas and Swift 1998), YAP was the first engine to actually integrate these two very different, and yet related, mechanisms into a single engine, OPTYAP (Rocha *et al.* 2005b). In our experience, the YAP tabling mechanisms are the most widely used extension of YAP, and are a key focus for the future of our system. Parallelism has been a less widely used feature of YAP, although our work in supporting parallelism was most beneficial in implementing the YAP *threads* library. Recent advances in computer architecture have rekindled interest in implicit parallelism in YAP (Santos Costa *et al.* 2010).

The paper is organized as follows. In Section 2, we first give a brief overview of the system history, adapted from (Santos Costa 2008). Next, in Section 3, we present the general structure and discuss the main data structures in the YAP engine. Section 4 presents the main contributions in the engine, and Section 5 discusses the design of the compiler. The two are tightly integrated (Santos Costa 1999). Section 6 discusses the implementation of the JITI, and Section 7 presents OPTYAP. We conclude by discussing some of the main issues in our work in Section 8, and present conclusions in Section 9. Throughout the text, we assume that the reader will have good familiarity with the general principles of Prolog implementation, and namely with the Warren Abstract Machine (WAM) (Warren 1983).

## 2 A little bit of history

The history of Prolog and Logic Programming starts in the early seventies, with the seminal works by Colmerauer, Roussel and Kowalski (Colmerauer 1993). The original Marseille Prolog was promptly followed by quick progress in the design of Logic Programming systems. One of the most exciting developments was David H. D. Warren's abstract interpreter, eventually called the WAM (Warren 1983), which became the foundation of Quintus Prolog (Quintus 1986). The success of Quintus Prolog motivated the development of several Prolog systems. YAP is one

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

example, and was started by Luís Damas and colleagues in 1984 at the University of Porto. Luís Damas had returned from the University of Edinburgh, where he had completed his PhD on type systems (Damas and Milner 1982). He was also interested in Logic Programming and, while at Edinburgh, had designed one of the first Prolog interpreters, written in the IMP programming language for the EMAS operating system, which would become the basis for the famous C-Prolog interpreter (Pereira 1987). Together with Miguel Filgueiras, who also had experience in Prolog implementation (Filgueiras 1984), they started work on the development of a new WAM-based Prolog. The goal was to design a compact, very fast system emulator, written in assembly. To do so, Luís Damas wrote the compiler in `C` and an emulator in `68000` assembly code.

Arguably, one of the strengths of YAP derives from Luís Damas' experience in Edinburgh: internal object representation was well defined from the start and always facilitated development. YAP included several improvements over the original WAM design: it used a depth-first design to visit terms, and was one of the first Prologs to do indexing on sub-terms (Santos Costa 1988). YAP also provided a very fast development environment, due to its `C`-written compiler. The combination of fast compilation and execution speed attracted a dedicated user community, mainly in Artificial Intelligence (e.g., Moniz Pereira's group supported the first YAP port to the VAX architecture). A major user was the European Union Eurotra project (Arnold *et al.* 1986) for which YAP developed *sparse functors*: one of the first attempts at using named fields for structures in Prolog.

The second chapter in YAP's history started on the mid-nineties. At this point in time, YAP development had slowed down. One problem was that the system had become very complex, mainly due to the need to support several instruction set architectures in assembly (at the time: 68000, VAX, MIPS, SPARC, HP-RISC, Intel x86). Unfortunately, a first attempt at using a `C` interpreter resulted in a much slower system. On the other hand, the user community was not only alive but growing, as Rui Camacho had taken YAP to the Turing Institute Machine Learning Group, where it was eventually adopted by Inductive Logic Programming (ILP) systems such as P-Progol, later Aleph (Srinivasan 2001), and IndLog (Camacho 1994). Second, researchers such as Vítor Santos Costa and Fernando Silva had returned to Porto and were interested in Parallel Logic Programming. While SICStus Prolog would have been an ideal platform, it was a closed-source system. YAP therefore became a vehicle of research first in parallelism (Rocha *et al.* 1999) and later in tabling (Rocha *et al.* 2005b). A new, fast, and `C`-based emulator was written to support this purpose (Santos Costa 1999) and brought YAP back to the list of the fastest Prolog systems (Demoen and Nguyen 2000).

The interest in YAP grew during the late nineties, leading to the third chapter in YAP's story. As hardware scaled up and users had more data to process, limitations in the YAP design become clear: Prolog programs perform well for small applications, but often just crash or perform unbearably slowly as application size grows. Effort has therefore been invested in rethinking the basics, step by step. The first step was rewriting the garbage collector (Castro and Santos Costa 2001). But the main

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

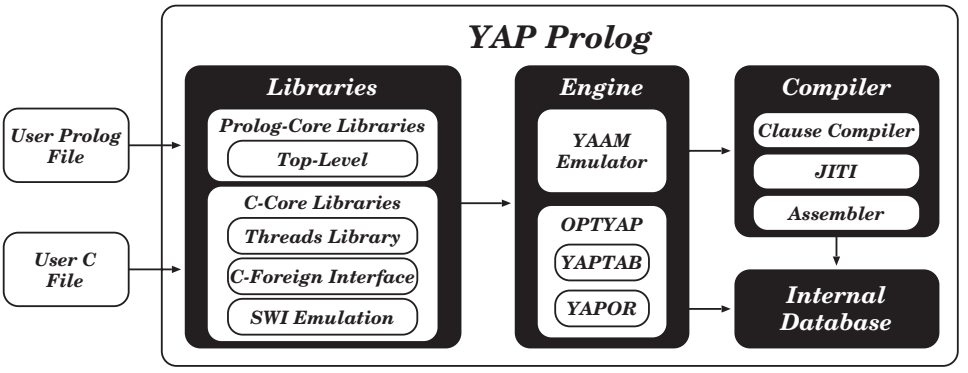



Fig. 1. The organization of the YAP Prolog system. At the heart of the system, we show the YAAM emulator with the OPTYAP extensions. The engine is supported by a core set of libraries written in C. These libraries can be user-extended through YAP's native foreign language interface and through SWI's interface. The compiler and JITI mechanisms are controlled by the engine and generate code to be stored in the internal database.

developments so far have been in indexing: it had become clear that the WAM's approach to indexing simply does not work for applications that need to manipulate complex and large databases. Just-In-Time indexing (Santos Costa *et al.* 2007) tries to address this problem.

## 3 System organization

Figure 1 presents a high-level view of the YAP Prolog system. The system is written in C and Prolog. Interaction with the system always starts through the top-level Prolog library. Eventually, the top level refers to the core C libraries. The main functionality of the core C libraries includes starting the Prolog engine, calling the Prolog clause compiler, and maintaining the Prolog internal database. The Prolog sequential engine executes YAP's YAAM instructions (Santos Costa 1999), and has been extended to support tabling and or-parallelism (Rocha *et al.* 2005b). The engine may also call the JITI (Santos Costa *et al.* 2007). Both the compiler and the JITI rely on an assembler to generate the code that is stored in the internal database.

The C-core libraries further include the parser and several built-ins (not shown in Fig. 1). An SWI-compatible threads library (Wielemaker 2003) provides the support to thread creation and termination, and access to locking. The Foreign Language Interface (FLI) library allows external C-code to use the YAP data structures. YAP also provides an SWI FLI emulator that translates SWI-Prolog's (Wielemaker 2010) FLI to YAP FLI calls. SWI-Prolog packages such as `chr` (Schrijvers 2008), `JPL`, `SGML` (Wielemaker 2010), and even the core SWI-Prolog Input/Output routines (`PLStream`) have been adapted to use this layer.

### 3.1 The key data structures

Throughout, the YAP implementation uses *abstract types* to refer to objects with similar properties, say, the type `Term` refers to all term objects. Each abstract type

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

may have different concrete types (or subtypes), but a concrete type has a single abstract type by default. For example, the abstract type `Term` has the concrete type `Appl` (compound term), `Pair` (lists), or `Int`. In all cases, given a subtype `T` of some abstract type `A`, the following three functions should be available:

- Given a concrete object of concrete type `T`, the *Abs*`T` routine returns an instance of its abstract type `A`.
- Given an instance of the abstract type `A`, the *Rep*`T` routine returns an object with concrete type `T`.
- Given some arguments, the constructor *Mk*`TA` constructs an object of concrete type `T` and returns an instance of `A`.

For example, given a pointer to the stack, the function *AbsAppl* returns a `Term` object; given an object of type `Term`, the function *RepAppl* returns a pointer to the stack; and lastly, the function *MkApplTerm* receives a functor, an arity, and an array of terms and returns an object of type `Term`. In order to achieve the efficiency, most of these functions are implemented as inline `C`-functions.

### 3.2 The database

YAP includes two main data structures: the Engine Context and the Database. The *Engine Context* maintains the abstract machine internal state, such as abstract registers, stack pointers, and active exceptions. The *Database* data structure maintains the root pointers to the internal database, including the *Atom Table* and the *Predicate Table*. The table is accessible from a root pointer so that the state of the engine can be *saved* to and *restored* from a dump file.

In order to support parallelism and threads, YAP organizes the database as:

- The `GLOBAL` structure that is available to all workers; locks should protect access to these data structures.
- An array of per-worker structures, where each one is called `LOCAL`. We define a worker to be a scheduling unit that can run an YAAM engine, that is, a thread or a parallel process. The engine abstract registers are accessible through the worker's `LOCAL`.

The structure of the database is presented in Figure 2. We assume the support for two workers; hence, we require two `LOCAL`. Notice that each `LOCAL` structure contains a copy of the YAAM registers. The main structure in the YAP database is the *Atom Table*, containing objects of abstract type `Atom`. The abstract type `Atom` has a single concrete type, `AtomEntry`. Thus, the Atom Table is implemented as a hash table with linked lists of `AtomEntry` objects. Each `AtomEntry` contains

(1) `NextOfAE`: a pointer to the next atom in the linked list for this hash entry;
(2) `PropsOfAE`: a pointer to a linked list of atom *properties*; the atom's string.

The `Prop` type abstracts the objects that we refer to by the atom's name. Example subtypes of `Prop` include the functors, modules, operators, global variables,

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

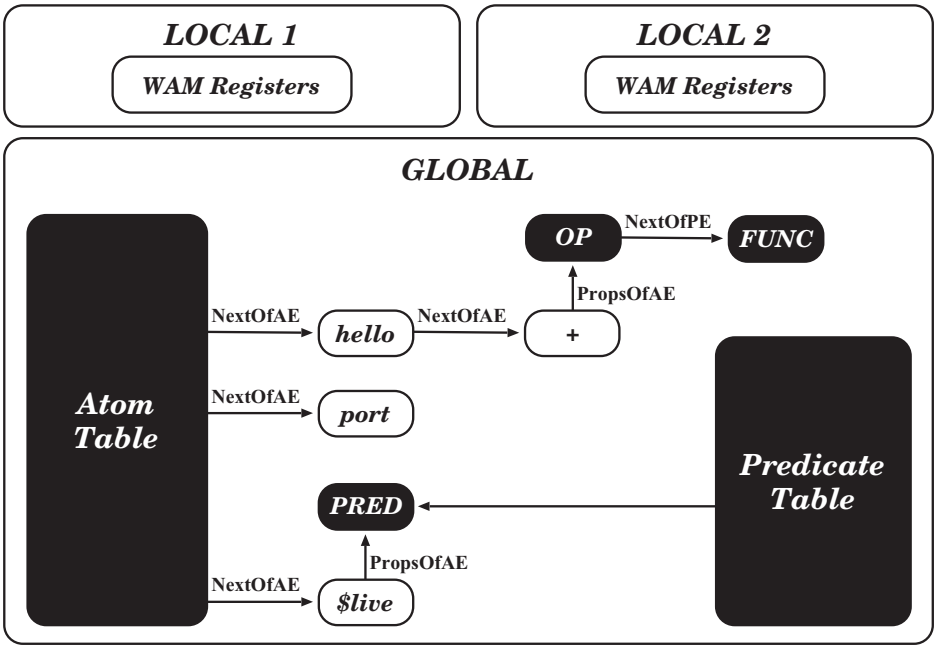



Fig. 2. Organization of the YAP database. Each worker has a local set of variables (including abstract machine registers). All workers share a `GLOBAL` data structure that includes a hash-based atom table. Atoms are stored in a linked list, and most of their properties, including predicates with the atom as name, are stored in a linked list for each atom. Predicates are often used, so there is a direct hash table for them.

blackboard entries, and predicates. All of them are available by looking up an atom and following the linked list of `Prop` objects.

Figure 2 shows an atom table with four atoms: `hello`, `+`, `port`, and `$live`. Notice that only `+` and `$live` have associated properties. In practice, most atoms do not have properties.

Every concrete type of `Prop` implements two fields:

(1) `NextOfPE` allows organizing properties for the same atom as a linked list;
(2) `KindOfPE` gives the type of property.

Each property extends the abstract property in its own way. As an example, *functors* add three extra fields: a back pointer to the atom, the functor's arity, and a list of properties.

This design is based on LISP implementations, and has been remarkably stable throughout the history of the system. Main optimizations and extensions include:

(1) Recent versions of YAP support two atom tables: one groups all ISO-Latin-1 atoms, where each character code $c$ is such that $0 < c < 255$, and the other stores atoms that need to be represented as wide strings. YAP implements two tables in order to avoid an extra field per atom.

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

(2) As discussed above, functors have their own `Prop` objects, namely, predicates and internal database keys with that functor. This was implemented to improve the performance of meta-calls.
(3) The case where we have predicates with the same functor but belonging to different modules is addressed by a *predicate hash table*, which allows direct access to a predicate from a functor module key.

In Figure 2, the atom `+` has two properties: one of the type `op` and another of type `functor`. The atom `$live` has a property of type `predicate`.

Traditionally, YAP allocates memory as a single big chunk and then uses its own memory allocator. This has two advantages: it avoids the overhead of going through the standard `C` library, and simplifies state saving and restoring. The current allocator is Doug Lea's global memory allocator (Lea 1996).

YAP can also use the `C`-library malloc, as a configuration time option. This is most useful in situations where YAP needs to share the memory with other systems (e.g, the Java interface).

### *3.3 Dynamic data structures*

Each worker (parallel process or thread) maintains four separate stacks and a set of registers. The stacks are:

- *Global Stack*: implemented as an array of `CELL`s, it stores abstract objects of type `Term`.
- *Local Stack*: implemented as an array of `CELL`s, it stores choice points and environments.
- *Trail*: it stores objects of type `TrailEntry`.
- *Auxiliary Stack* (generalizes the WAM PDL Warren 83): a pointer to a reusable area of memory used to store temporary data, such as the stacks used for unification or other term matching operations.

Objects of type `Term` reside in the Global and Local Stacks, and are always constructed from `CELL`s. YAP defines six concrete types:

(1) *Small integers* are constructed from a subset of type `Int` and occupy a single cell, with up to 29 bits in 32 machines. `Int` is an integer-like type defined to take the same space as `CELL`.
(2) *Atoms* occupy a single cell and are constructed from the objects of the abstract type `Atom`.
(3) *Applications*, or compound terms, occupy $N + 1$ cells, where the first cell is reserved for an object of abstract type `Functor` with arity $N$, and the remaining $N$ cells for objects of type `Term`.
(4) *Pairs*, occupy 2 cells, where the first cell is a `Term` called the *Head*, and the second a `Term` called the *Tail*.
(5) *References* occupy 1 cell, and are pointers to objects of type `Term`. By default, YAP represents free variables as self-references, but it can support free variables as the `NULL` pointer.

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

(6) *Extensions* occupy $N+2$ cells: the *header*, a variable number $N$ of cells, and the *footer*. The engine understands three extensions: floating point numbers, large integers, and *blobs*, originally introduced to support very large integers.

### 3.4 Tagging scheme

Each different concrete type should have its own *tag*. Tag schemes differ significantly between Prolog systems (Schimpf 1990; Tarau and Neumerkel 1994); we refer the reader to (Morales *et al.* 2008) for a recent investigation of this issue. The YAP tag scheme was designed to be efficient (Santos Costa 1999), and to allow using the whole available memory in 32 bit machines. This allows at most 2 bits for tags.

Unfortunately, there are six different concrete types: this would require $\lceil log_2 6 \rceil = 3$ bits, but in order to access the full address space, we are constrained to the two lower bits. The solution was:

- Atoms and small integers share the same tag. YAP allocates each atom as a separate object, guaranteeing that the object is always allocated at an address multiple of 8, so that the third lowest bit can be used to distinguish between the two cases.
- Applications and extensions share the same tag. The header of an extension is a small number. This number is guaranteed to be an invalid address in modern systems, as these systems never allocate memory on the first virtual memory page.

This scheme allows for taking advantage of all the available memory with a 32 bit `CELL`, but slows down access to compound terms. A second drawback is that it requires explicit code for efficiency, making it hard to take advantage of 64 bit machines. Notice that YAP does not need tag bits for garbage collection: instead, we use a separate memory area to store the garbage collector state (Vandeginste and Demoen 2007).

Blobs were initially introduced to support very large numbers. They enhance YAP functionality without requiring extensive changes to the engine, and currently provide the following functionality:

- `BIG_INT`: very large integers, currently implemented through an interface to the GMP package (Granlund 2004).
- `BIG_RATIONAL`: rationals.
- `STRING`: sequences of character codes.
- `EMPTY_ARENA`: a chunk of cells that can be used to construct global variables or global data structures. This is used to support `nb_` predicates and to implement `findall/3` and the `nb` library of global queues, global heaps, and global beams.
- `ARRAY_INT`: a multidimensional array of (non-tagged) integer numbers. This is manipulated by the package `matrix` .
- `ARRAY_FLOAT`: a multidimensional array of (non-tagged) floating-point numbers. This is manipulated by the package `matrix`.
- `CLAUSE_LIST`: a sequence of pointers to code. This allows for dynamic choice points, and is used by the user-defined indexers (Vaz *et al.* 2009).

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

## 4 The engine

YAP implements a version of David H. D. Warren's WAM (Warren 1983). Other Prolog systems using the WAM include SICStus Prolog (Carlsson and Widen 1988), Ciao Prolog (Hermenegildo *et al.* 2008), XSB (Sagonas and Swift 1998), GNU Prolog (Diaz and Codognet 2001), and $ECL^iPS^e$ (Aggoun *et al.* 1995). The original machine consisted of 33 instructions used to implement an environment-based term-copying strategy. WAM instructions were divided into:

- Argument unification, or `get` and `unify` instructions;
- Argument building, or `put` instructions;
- Control: `call`, `execute`, `proceed`, `allocate`, and `deallocate`;
- Choice-point manipulation, or `try` instructions;
- Indexing, or `switch` instructions;
- Cut instructions.

The WAM instructions are very well suited to compilation: one compiles a term by walking depth-first and left to right and associating each symbol with an operation. Arguably, the WAM performs quite well and is very well understood. On the other hand, most decisions on the WAM were taken a long time ago, and there has been recent interest in other abstract machine architectures for Prolog (Zhou 2007).

YAP implements the YAAM emulator as a large `C` function. The `C`-code for each instruction always starts with an `Op` macro and always terminates with an `EndOp` macro. Since YAP-6, a Prolog program, `buildops`, understands these macros and uses them to generate a file with all the YAAM opcodes, required by the assembler, and a file with commands to restore a clause or to be executed when the atom garbage collector needs to walk over YAAM instructions.

The emulator initializes by copying YAAM registers to local storage. Whether this datum is in the call stack or in the registers depends on the Instruction Set Architecture, Operating Systems, Compiler, and functionality being supported (Santos Costa 1999). YAP then starts executing YAAM instructions. Next, we discuss the main differences between the YAAM (Santos Costa 1999) and the WAM.

### *4.1 Unification instructions*

There are several interesting issues regarding unification instructions. A first problem is whether we should globalise void variables occurring in the body of a clause. Consider the following code fragment:

```
a(X) :- b(X,Y), c(X,Z).
```

The variable `Y` is a void variable, and can be compiled either as a `put_x_var` instruction, or as a `put_y_var`. Compiling it as `put_x_var` requires placing the void variable in the Global Stack: thus, space allocated to this variable can only be recovered through backtracking, or through garbage collection. Compiling it as `put_y_var` requires placing the variable in the Local Stack, and space can be recovered as soon as we call `c/2`. We have experimented with both approaches, and rarely noticed significant differences. YAP traditionally follows the first approach, mostly

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

in order to simplify compilation. Notice that systems such as BIM-Prolog (Mariën 1993), Aquarius (Van Roy 1990), or hProlog (Demoen and Nguyen 2000) address this problem simply by globalizing all free variables..

A second problem is how to support nested unification (Mariën and Demoen 1991). Consider the clause

```
a([X,f(Y,X),Y]).
```

The WAM compiles the term breadth-first, obtaining:

```
a([X|Z]) :- Z = [A121|A122], A121 = f(Y,X), A122 = [Y].
```

Notice that the WAM approach requires extra temporary variables. SICStus Prolog optimises the specific case of lists through the `unify_list` instruction that follows a list depth-first (Carlsson 1990).

YAP implements a more general solution to this problem, first published by Meier (Meier 1990). Sub-terms are always compiled depth-first to `unify_` instructions. Thus, YAP will generate the following code in this case:

```
get_list        A1              pop             1
unify_var       X1              unify_last_list
unify_last_list                 unify_val       X2
unify_struct    f/2             unify_last_atom []
unify_var       X2              proceed
unify_last_val  X1
```

The code assumes an unification stack, initialized by the `get_list` and `get_struct` instructions. Each `unify_list` or `unify_struct` instruction pushes the current state into the stack. The `pop` instruction pops this state if necessary.

This algorithm is straightforward to implement; it does not put pressure on temporary registers; and it allows inheritance of modes. If YAP enters a structure in `write`-mode, then all its sub-structures will execute in `write`-mode.

A corollary of this advantage is that `write` code performs less tests, and is therefore simpler. This observation motivated the double-opcode scheme originally presented in (Santos Costa 1999). In this method, each `unify` instruction has two opcodes: one taken in `read` mode and the other taken in `write` mode. It can be easily shown that every `write-unify` instruction is followed by:

- a `write-unify` instruction, if we are executing within the same sub-term;
- a `pop` instruction, if we are moving to the parent sub-term;
- a `get` or `control` instruction, if we have exited the term.

A similar argument can be made for most `read-unify` instructions, with the exception of `unify_list` and `unify_struct`. Therefore,

- `unify_list` and `unify_struct` instructions either preserve `write` mode, or may move from `write` to `read` mode;
- all other `unify_` instructions preserve `write` mode;
- `pop` instructions restore the `read/write` mode from the unification stack.

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

In our experience, separating write and read opcodes results in both a faster and a cleaner engine implementation.

Other interesting design issues for compound terms include:

- In `get_struct` and `unify_struct` instructions, YAP immediately initializes the arguments of the newly generated compound term as unbound variables. YAP uses this eager approach because it be can be implemented as a tight loop, improving locality, and because it allows discarding `unify_void` instructions at the end of a compound term.
- YAP uses `unify_last` instead of `unify` for the last instruction of a compound term. The `unify_last` instructions do not need to update the `S` register, simplifying code. Moreover, `unify_last_list` and `unify_last_struct` instructions do not need to push the current state to the unification stack.
- YAP completes a sub-term when executing `unify_last_atom`, `unify_last_var` or `unify_last_val`. Next, YAP may need to execute a `pop` instruction to return to a sub-term above. The `pop` instruction will then set the read/write mode by choosing the read or write opcode of the next instruction.

### *4.2 Term sharing*

Consider the following clause:

```
a(X,W,[Y,Z]) :- b([Y,Z]), a(W,f([Y,Z])).
```

The standard WAM would create three copies of `[Y,Z]`: one for the head-term and two for the body terms. YAP instead generates the following code:

```
get_var          Y1,A2          allocate
get_list         A3             call         user:b/1,1
save_pair        Y0             put_val      Y1,A1
unify_var        X0             put_struct   f/1,A2
unify_last_list                 write_val    Y0
unify_var        X0             deallocate
unify_last_atom  []             execute      user:a/2
put_val          Y0,A1
```

The `save_pair` instruction stores *AbsPair*(`S`), where `S` refers to the WAM's `S` structure pointer register, in an abstract machine register or environment slot. This argument is then stored in `Y0` in lieu of the term. This has two advantages:

- Increased sharing, while reducing code size and run-time memory overhead;
- Reduce the number of permanent variables. In the example, variables `Y` and `Z` are made void by this optimization. In contrast, the WAM would mark them as permanent variables.

The compiler maintains a table with all terms compiled so far in order to support this operation. Every time a copy is found the term is replaced by the argument. Notice that compilation times may increase on large clauses with many terms. Thus,

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

YAP imposes a maximum threshold on the number of terms that can be considered for reuse.

In a related optimization, recent versions of YAP compile *large* ground terms offline. That is, the clause

```
a("Long String").
```

is compiled as:

```
get_dbterm  [76,111,110,103,32,83,116,114,105,110,103],A1
proceed
```

Currently, `get_dbterm` simply unifies its argument with a ground term in the database. This has two advantages: it reduces code size and makes string construction constant time. The major drawbacks are the cost of maintaining an extra database of terms and the need to implement JITI support.

### *4.3 Non-logical features*

Actual Prolog implementations must support non-logical features such as the cut, disjunctions, and type predicates. YAP always stores a cut pointer in the environment (Mariën and Demoen 1989). The implementation of disjunction is more complex. Two basic approaches are (Carlsson 1990; Demoen *et al.* 2000):

- *Offline* compilation (Carlsson 1990) generates a new intermediate predicate and compiles disjuncts as new clauses. It allows for simpler compilation.
- *Inline* compilation uses special instructions to implement disjunction (Demoen *et al.* 2000). It can reduce overheads.

YAP implements inline compilation of disjunctions. Each clause is divided into a graph where an edge is an alternative to a disjunction, and each edge starts with an `either` an `or_else`, or `or_last` instruction. These instructions implement a choice point with arity 0, as all shared variables are guaranteed to the environment.

As most other Prolog compilers, YAP also inlines a number of built-ins (Nässén *et al.* 2001; Zhou 2007):

(1) Type built-ins such as `var`, `nonvar`, `atom` and related. They are implemented as `p_var` , `p_nonvar` , `p_nonvar` instructions.
(2) Arithmetic operations. Currently, YAP only optimises the integer operations. Examples include the `p_plus` instructions, which are further optimised according to whether one of the arguments is a constant or not.
(3) The `functor` and `arg` built-ins. YAP implements different `functor/3` instructions, depending on how arguments were instantiated at compile time.
(4) The meta-call: YAP inlines some meta-calls (Tronçon *et al.* 2007). This is difficult due to the complexity of the `goal_expansion` and the module mechanism.

The implementation of inline built-ins has overgrown the initial design, and requires redesign and a cleanup.

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

## 5 Compilation

The YAP compiler implements the following steps algorithm:

(1) `c_head`: generate a WAM-like representation for the head of the clause.
(2) If the clause is a ground fact, proceed to step (6).
(3) `c_body`: generate WAM-like representation for the body of the clause.
(4) `c_layout`: perform variable classification and allocation.
(5) `c_optimize`: eliminate superfluous instructions.
(6) Assemble the code and copy it to memory.

The `c_head` step simply walks over the clause head and generates a sequence of WAM instructions. The `c_body` routine visits the body goals and generates code for each goal in sequence. Special care must be taken with disjunctions and inline built-ins.

Both `c_head` and `c_body` call `c_goal` to generate code for the head and sub-goals. The main challenge is to compile variables, performed by `c_var`. Each variable is made to point to a `VarEntry` structure, that contains, among other information: (i) a reference count indicating how many times the variable was used in the clause; (ii) the first occurrence of the variable in the code; and, (iii) the last occurrence. The `c_var` routine then works as follows:

- If this is the first occurrence of the variable, *bind* the variable to a `VarEntry`, set to have a reference count to 1, and set the first and last occurrence to the current position.
- Otherwise, increment reference count and set the last occurrence to the current position.

`c_var` must also generate a WAM-like instruction for the variable. It generates a `unify` instruction for variables in sub-terms; a `put` instruction for variables in the body of the clause; a `get` instruction for variables in the head.

The `c_layout` routine proceeds as follows:

(1) Reverse the chain of instructions.
(2) Going from the end to the beginning, check if a variable must be permanent, and if so give it the next available environment slot. This guarantees that the environment variables occurring in the rightmost goals have the lower slots. This step again reverses the chain.
(3) Going from the beginning to the end, allocate every temporary variable using a first-come, first-served greedy allocation algorithm. The YAAM has a very large array of registers, and spilling is considered an overflow.

The `c_optimize` step searches for unnecessary instructions, say, `get_x_val A1,X1` and removes them.

### *5.1 Compiling disjunctions*

A clause with disjunctions can be understood as a directed acyclic graph. Each node in the graph either delimits the beginning/end of the clause or the beginning/end

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

of a disjunction. Edges link nodes that delimit an or-branch. Notice that there is always an edge that includes the head of the clause; we shall name this edge the *root-edge*. Thus, a Horn Clause has two nodes and a single edge, whereas a clause of the form

```
a :- (b ; c,d), e
```

has four nodes and four edges. YAP uses the following principles to compile the disjunctions:

- Any variable that crosses over two edges has to be initialized in the *root-edge*. This prevents dangling variables, say:

  ```
  g :- ( b(X) ; c(Y) ), d(Y).
  ```

  The `Y` variable may be left dangling if not initialized before the edge.
- As usual, environments are allocated if there is a path in the graph with two user-defined goals, or a user-defined goal followed by built-ins.
- If a disjunction is of the form $G \rightarrow B_1; B_2$ and $G$ is a conjunction of *test* built-ins, the compiler compiles $G$ with a jump to a *fail* label that points to $B_2$.
- Otherwise, the compiler generates choice-point manipulation instructions: the `either` instruction starts the disjunction; the `or_else` for inner edges; and the `or_last` prepares the last edge for the disjunction.

There are cases where YAP has to do better. Consider a *fast* implementation of fibonacci:

```
fib(N, NX) :- ( N =< 1 ->
                  NX = 1
                ;
                  N1 is N - 1, N2 is N - 2,
                  fib(N1, X1), fib(N2, X2),
                  NX is X1 + X2
                ).
```

The variables `N` and `NX` cross the disjunction; therefore, the above algorithm initializes them as permanent variables at the root edge. The problem is that the YAP variable allocator will use the environment slots to access `N` and `NX`, and would fail to take advantage of the fact that an `N` is available in $A_1$ and `NX` in $A_2$. This generates unnecessary accesses and the code may be less efficient than creating a choice point and executing a separate first clause. The solution is to delay environment initialization until one is sure one needs it. The rules are:

- Environments are allocated only *once*: the edge that allocates the environment is the leftmost–topmost edge $E$ such that

1. no edge $E'$ above needs an environment,
2. no edge to the left of $E$ needs the environment,

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

3. $E$ or a descendant of $E$ needs the environment, and
4. at least a descendant of a right sibling of $E$ needs the environment.

- Variables are copied to the environment after allocation.

Applying these rules allows the compiler to delay marking some variables as permanent variables. This simplifies the task of the variable allocator, and leads to much faster code in the case above.

### 5.2 The assembler

The YAP Prolog assembler converts from a high-level representation of YAAM instructions into YAAM byte code. It executes in two steps:

(1) Compute the addresses for labels and perform peephole optimizations, such as instruction merging.
(2) Given the addresses of labels, copy instructions to actual location.

Instruction merging (Santos Costa 1999; Demoen and Nguyen 2000; Nässén *et al.* 2001; Zhou 2007) is an important technique to improve the performance of emulators. The assembler implements instruction merging:

(1) where it leads to improvement of the performance in recursive predicates: examples include `get_list` and `unify_x_val`, or `put_y_val` followed by `put_y_val`;
(2) where it leads to substantial improvements in code size: examples include sequences of `get_atom` instructions that are typical of database applications (Santos Costa 2007).

## 6 The JITI

YAP includes a JITI (Santos Costa *et al.* 2007; Santos Costa 2009). Next, we give a brief overview of how the algorithm has been implemented in the YAP system. First, we observe that in YAP, in contrast to the WAM, by default, predicates have *no* indexing code. All indexing is constructed at run time.

Our first step is thus to ensure that calls to non-indexed predicates have the abstract machine instruction `index_pred` as their first instruction. This instruction calls the function `Yap_PredIsIndexable` that implements the JITI.

### 6.1 The indexing algorithm

Indexing has been well-studied in Prolog systems (Carlsson 1987; Demoen *et al.* 1989; Van Roy 1990; Zhou *et al.* 1990). The main novelty in the design of the JITI is that it tries to generate the code that is well-suited to the instantiations of the goal. To do so, it basically follows a decision tree algorithm, where decisions are made by inspecting the instantiation of the current call. The actual algorithm is as follows:

(1) Store the pointers to every clause in the predicate in an array `Clauses` and compute the number of clauses.

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

(2) Call `do_index(Clauses,1)`, where the number 1 refers to the first argument.
(3) Assemble the generated code.

The function `do_index` is the core of the JITI. It is a recursive function that, given a set of clauses $C$ with size $N$ and an argument $i$, works as follows:

(1) If $N \leqslant 1$, call `do_var` to handle the base case.
(2) If $i > Arity$, we have tried every argument in the head: call `do_var` to generate a `try-retry-trust` chain.
(3) If $A_i$ is unbound, first call `suspend_index(Clauses,i)`, to mark this argument as currently unindexed, and then call `do_index(Clauses,i+1)`.
(4) Extract the constraint that each clause $C$ imposes on $A_i$, and store the constraint in $Clauses[C]$. The YAP JITI understands two types of constraints:
   - bindings of the form $X = T$, where the main functor of $T$ is known;
   - type constraints, such as $number(X)$.
(5) Compute the *groups*, where a group is a contiguous subset of clauses that can be indexed through a single `switch_on_type` (Warren 1983). For example, consider the following definition of predicate `a/1`:

```
a(1). a(1). a(2). a(X). a(1).
```

   This predicate has three groups: the first three clauses form a group, and the fourth and fifth clauses form each one a different group. The fourth clause forms a *free* group, as it imposes no constraint on $A_1$.
(6) In order to generate simpler code, if the number of groups $NG$ is larger than one and we are not looking at the first argument, that is, $NG > 1 \wedge i > 1$, then do not try indexing the current argument, and instead call `do_index(Clauses,i+1)`.
(7) Compile the groups one by one. If the group is free, call `do_var`: this function generates the leaf code for a sequence of `try-retry-trust` instructions.
   Otherwise, if all constraints in the group are binding constraints:
   (a) Generate a `switch_on_type` instruction for the current argument $i$.
   (b) The `switch_on_type` instruction has four slots in the YAAM (and in the WAM): constants, compound terms, pairs, and unbound variables. The JITI generates the code for the first three cases. The fourth case is not compiled; instead, the JITI fills the last slot with the *expand_index* instruction (discussed in detail later).
   (c) Next, separate clauses in three subgroups according to whether they contain a constant (atoms or small integers), a pair, or a compound term, including extensions.
   (d) Call `do_consts`, `do_funcs`, and `do_pair` on each subgroup to fill in the remaining slots.

   A clause imposing a type constraint requires specialized processing, for example:
   (a) $integer(A_i)$ adds the clause to the list of constants and the list of functors.
   (b) $var(A_i)$ requires removing the current clause from the list of constants, functors and pairs;
   (c) $nonvar(A_i)$ cannot select between different cases, and is not used.

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

The do_var auxiliary routine is called to handle the cases where we cannot index further: it either commits to a clause, or creates a chain of try-retry-trust instructions. The do_consts, do_funcs, and do_pair functions try to construct a decision list or hash table on the values of the main functor of the current term, in a fashion very similar to the standard WAM. On the other hand, do_funcs, and do_pair may call do_compound_index to index on sub-terms. Finally, YAP implements a few optimizations to handle common cases that do not fit well in this algorithm (e.g., catch-all clauses).

The suspend_index(Clauses,i) function generates an expand_index $A_i$ instruction at the current location, and then continues to the next argument. At run time, if ever the instruction is visited with $A_i$ bound, YAP will expand the index tree, as discussed next.

### 6.2 Expanding the index tree

The expand_index YAAM instruction verifies whether new calls to the indexing code have the same instantiation as the original call. Thus, it allows the YAP JITI to grow the tree whenever we receive calls with different modes. The instruction executes as follows. First, it recovers the *PredEntry* for the current predicate, and then it calls Yap_ExpandIndex that proceeds as follows:

(1) Initialize clause and groups information.
(2) Walk the indexing tree from scratch, finding out which instruction caused the transfer to expand_index, and what clauses matched at that point. Store the matching clauses in the *Clauses* array.
(3) Call do_index(Clauses, i+1) to construct the new tree.
(4) Link the new tree back to the current indexing tree.

The second step is required because when we call expand_index, we do not actually have a pointer to the previous instruction, nor do we know how many clauses do match at this point (doing so would very much increase the size of the indexing code). Instead, we have to follow the indexing code again from scratch. As Yap_ExpandIndex executes each instruction in the indexing tree, it also selects the clauses that can still match. The algorithm is as follows:

(1) Set the alternative program pointer, $AP$ to NULL, the parent program pointer $P'$ to NULL, and the program pointer $P$ to point at the initial indexing instruction.
(2) While the YAAM instruction expand_index was not found:
(3) Set the current instruction pointer $P$ to be $P'$.
(4) If the current opcode is:
    - switch_on_type, then check the type of the current argument $i$, remove all clauses that are constrained to a different type from *Clauses*, and compute the new $P$.
    - switch_on_{cons, struct}, then check if the current argument $i$ matches one of the constants (functors). If so, remove all clauses that are constrained to a different constant from *Clauses*, and take the corresponding entry. If not, jump to $AP$.

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

- `try`, then mark that we are not the first clause and set $AP$ to the next instruction.
- `retry`, then set $AP$ to be the next instruction and jump to the label.
- `trust` set then $AP$ to `NULL` and jump to the label.
- `jump_if_nonvar`, then check if the current $A_i$ is bound. If not, proceed to the next instruction. Otherwise, if the jump label is `expand_index`, we are done.

The algorithm returns a set of clauses *Clauses* and a pointer $P'$ giving where the code was called from. We thus can call `do_index` as if it had been called from the `index_pred` instruction.

### 6.3 The JITI: discussion

The main advantages of the JITI are the ability to index multiple arguments and compound terms, and the ability to index for multiple modes of usage. Several Prolog systems do support indexing on multiple arguments (Wielemaker 2010; Zhou 2007; Sagonas *et al.* 1997); on the other hand, we are not aware of other systems that allow multiple modes. Our experience has shown that this feature is very useful in applications with large databases. A typical example is where we use the database to represent a graph and we want to walk edges in *both* directions; a second typical application is when mining databases (Fonseca *et al.* 2009). Arguably, a smart programmer will be able to address these problems by duplicating the database: the JITI is about not having to do the effort.

The JITI has a cost. First, the index size can grow significantly, and, in fact, exceed the size of the original database (Fonseca *et al.* 2009). In the worst case, we can build a large index that will serve a single call. Fortunately, our experience has shown this to be rare. In most cases, if the index grows, it is because it is needed, and the benefits in running-time outweigh the cost in memory space. A second drawback is the cost of calling `Yap_ExpandIndex`. Although we have not systematically measured this overhead, in our experience, it is small.

## 7 OPTYAP: an overview

One of the major advantages of Logic Programming is that it is well-suited for parallel execution. The interest in the parallel execution of logic programs mainly arose from the fact that parallelism can be exploited *implicitly*, that is, without input from the programmer to express or manage parallelism, ideally making Parallel Logic Programming as easy as Logic Programming.

On the other hand, the good results obtained with tabling (Sagonas *et al.* 1997) raise the question of whether further efficiency would be achievable through parallelism. Ideally, we would like to exploit maximum parallelism and take maximum advantage of current technology for tabling and parallel systems. Towards this goal, we proposed the *Or-Parallelism within Tabling* ($OPT$) model. The OPT model generalizes Warren's multi-sequential engine framework for the exploitation of or-parallelism in shared-memory models. It is based on the idea that all open

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

alternatives in the search tree should be amenable to parallel exploitation, be they from tabled or non-tabled subgoals. Furthermore, the OPT model assumes that tabling is the base component of the parallel system, that is, each *worker* is a full sequential tabling engine, the or-parallel component only being triggered when workers run out of alternatives to exploit.

OPTYAP implements the OPT model, and we shall use the name OPTYAP to refer to YAP plus tabling and or-parallelism (Rocha *et al.* 2005b). OPTYAP builds on the YAPOR (Rocha *et al.* 1999) and YAPTAB (Rocha *et al.* 2000) work. YAPOR was previous work on supporting or-parallelism over YAP (Rocha *et al.* 1999). YAPOR is based on the environment copying model for shared-memory machines, as originally implemented in Muse (Ali and Karlsson 1990). YAPTAB is a sequential tabling engine that extends YAP's execution model to support tabled evaluation for definite programs. YAPTAB's implementation is largely based on the seminal design of the XSB system, the SLG-WAM (Sagonas and Swift 1998), but it was designed for eventual integration with YAPOR. Parallel tabling with OPTYAP is implicitly triggered when both YAPOR and YAPTAB are enabled.

### *7.1 The sequential tabling engine*

Tabling is about storing intermediate answers for subgoals so that they can be reused when a *variant call*[1] appears during the resolution process. Whenever a tabled subgoal is first called, a new entry is allocated in an appropriate data space, the *table space*. Table entries are used to collect the answers found for their corresponding subgoals. Moreover, they are also used to verify whether calls to subgoals are variant. Variant calls to tabled subgoals are not re-evaluated against the program clauses, instead they are resolved by consuming the answers already stored in their table entries. During this process, as further new answers are found, they are stored in their tables and later returned to all variant calls. Within this model, the nodes in the search space are classified as either: *generator nodes*, corresponding to first calls to tabled subgoals; *consumer nodes*, corresponding to variant calls to tabled subgoals; or *interior nodes*, corresponding to non-tabled subgoals.

To support tabling, YAPTAB introduces a new data area to the YAP engine, the *table space*, implemented using *tries* (Ramakrishnan *et al.* 1999); a new set of registers, the *freeze registers*; an extension of the standard trail, the *forward trail*; and four new operations for tabling. The configuration macro `TABLING` defines when tabling support is enabled in YAP. The new tabling operations are:

- The *tabled subgoal call* operation checks if a subgoal is a variant call. If so, it allocates a consumer node and starts consuming the available answers. If not, it allocates a generator node and adds a new entry to the table space. Generator and consumer nodes are implemented as standard choice points extended with an extra field, `cp_dep_fr`, that is a pointer to a *dependency frame* data structure used by the *completion* procedure. Generator choice

[1] Two calls are said to be variants if they are the same up to variable renaming.

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

points include another extra field, `cp_sg_fr`, that is a pointer to the associated *subgoal frame* where tabled answers should be stored. Tabled predicates defined by several clauses are compiled using the `table_try_me`, `table_retry_me`, and `table_trust_me` WAM-like instructions in a manner similar to the generic `try_me`/`retry_me`/`trust_me` WAM sequence. The `table_try_me` instruction extends the WAM's `try_me` instruction to support the tabled subgoal call operation. The `table_retry_me` and `table_trust_me` differ from the generic WAM instructions in that they restore a generator choice point rather than a standard WAM choice point. Tabled predicates defined by a single clause are compiled using the `table_try_single` WAM-like instruction, a specialized version of the `table_try_me` instruction for deterministic tabled calls.

- The *new answer* operation checks whether a newly found answer is already in the table, and if not, inserts the answer. Otherwise, the operation fails. The `table_new_answer` instruction implements this operation.
- The *answer resolution* operation checks whether extra answers are available for a particular consumer node and, if so, consumes the next one. If no answers are available, it suspends the current computation and schedules a possible resolution to continue the execution. It is implemented by the `table_answer_resolution` instruction.
- The *completion* operation determines whether a subgoal is completely evaluated and when this is the case, it closes the subgoal's table entry and reclaims stack space. Otherwise, control moves to one of the consumers with unconsumed answers. The `table_completion` instruction implements it. On completion of a subgoal, the strategy to implement answer retrieval consists in a top–down traversal of the completed answer tries and in executing dynamically compiled WAM-like instructions from the answer trie nodes. These dynamically compiled instructions are called *trie instructions* and the answer tries that consist of these instructions are called *compiled tries* (Ramakrishnan *et al.* 1999).

Completion is hard because a number of subgoals may be mutually dependent, thus forming a *Strongly Connected Component* (or *SCC*) (Tarjan 1972). The subgoals in an SCC are completed together when backtracking to the *leader node* for the SCC, i.e., the youngest generator node that does not depend on older generators. YAPTAB innovates by considering that the control of completion detection and scheduling of unconsumed answers should be performed through the data structures corresponding to variant calls to tabled subgoals, and does so by associating a new data structure, the *dependency frame*, to consumer nodes. Dependency frames are used to efficiently check for completion points and to efficiently move across the consumer nodes with unconsumed answers. Moreover, they allow us to eliminate the need for a separate completion stack, as used in SLG-WAM's design, and to reduce the number of extra fields in tabled choice points. Dependency frames are also the key data structure to support parallel tabling in OPTYAP.

Another original aspect of the YAPTAB design is its support for the dynamic mixed-strategy evaluation of tabled logic programs using batched and local

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

scheduling (Rocha *et al.* 2005a), that is, it allows one to modify at run time the strategy to be used to resolve the subsequent subgoal calls of a tabled predicate. At the engine level, this includes minor changes to the tabled subgoal call, new answer and completion operations, all the other tabling extensions being commonly used across both strategies.

More recent contributions to YAPTAB's design include the proposals to efficiently handle incomplete and complete tables (Rocha 2006). Incomplete tables are a problem when, as a result of a pruning operation, the computational state of a tabled subgoal is removed from the execution stacks before being completed. In such cases, we cannot trust the answers from an incomplete table because we may loose part of the computation. YAPTAB implements an approach where it keeps incomplete tables around and whenever a new variant call for an incomplete table appears, it first consumes the available answers, and only if the table is exhausted, it will restart the evaluation from the beginning. This approach avoids re-computation when the already stored answers are enough to evaluate the variant call. On the other hand, complete tables can also be a problem when we use tabling for applications that build arbitrarily many large tables, quickly exhausting memory space. In general, we will have no choice but to throw away some of the tables (ideally, the least likely to be used next). YAPTAB implements a memory management strategy based on a *least recently used* algorithm for the tables. With this approach, the programmer can rely on the effectiveness of the memory management algorithm to completely avoid the problem of deciding what potentially useless tables should be deleted.

Performance results for YAPTAB have been very encouraging from the beginning. Initial results showed that, on average, YAPTAB introduces an overhead of about 5% over standard Yap when executing non-tabled programs (Rocha *et al.* 2000). For tabled programs, results indicated that we successfully accomplished our initial goal of comparing favorably with current state-of-the-art technology since, on average, YAPTAB showed to be about twice as fast as XSB (Rocha *et al.* 2000). In more recent studies, by comparing YAPTAB with other tabling Prolog systems, the previous results were confirmed and YAPTAB showed to be, on average, twice as fast as XSB and Mercury (Somogyi and Sagonas 2006) and more than twice faster than Ciao Prolog and B-Prolog (Chico *et al.* 2008). Regarding the overhead for supporting mixed-strategy evaluation, our results showed that, on average, YAPTAB is about 1% slower when compared with YAPTAB supporting a single scheduling strategy (Rocha *et al.* 2005a). Moreover, our results showed that dynamic mixed strategies, incomplete tabling, and table memory recovery can be extremely important to improve the performance and increase the size of the problems that can be solved for ILP-like applications (Rocha 2007). Considering that YAP is one of the fastest Prolog engines currently available, these results are quite satisfactory and they show that YAPTAB is a very competitive tabling system.

### 7.2 The or-parallel tabling engine

In OPTYAP, or-parallelism is implemented through copying of the execution stacks. More precisely, we optimize copying by using *incremental copying*, where workers

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

only copy the differences between their stacks. All other YAP areas and the table space are shared between workers. Incremental copying is part of YAPOR's engine.

A first problem that we had to address in OPTYAP was concurrent access to the table space. OPTYAP implements four alternative locking schemes to deal with concurrent accesses to the table space data structures, the *Table Lock at Entry Level* (TLEL) scheme, the *Table Lock at Node Level* (TLNL) scheme, the *Table Lock at Write Level* (TLWL) scheme, and the *Table Lock at Write Level-Allocate Before Check* (TLWL-ABC) scheme. The TLEL scheme includes a single lock per trie, and thus, allows a single writer per trie. The TLNL has a lock per node, and thus, allows a single worker per chain of sibling nodes that represent alternative paths from a common parent node. The TLWL scheme is similar to TLNL, but the common parent node is only locked when writing to the table is likely. Lastly, the TLWL-ABC is an optimization that allocates and initializes nodes that are likely to be inserted in the table space before any locking is performed. Experimental results (Rocha *et al.* 2002) showed that TLWL and TLWL-ABC present the best speedup ratios and that they are the only schemes showing good scalability.

A second problem was *public completion*. When a worker $W$ reaches a leader node for an SCC $S$ and the node is public, other workers can still influence $S$, for example, if finding new answers for consumers in $S$. In such cases, $W$ cannot complete but, on the other hand, it would like to move elsewhere in the tree to try other work. Note that this is the only case where or-parallelism and tabling conflict. One solution would be to disallow movement in this case. Unfortunately, we would severely restrict the parallelism. As a result, in order to allow $W$ to continue execution, it becomes necessary to *suspend the SCC* at hand. Suspending an SCC consists of saving the SCC's stacks to a proper space and leave in the leader node a reference to the suspended SCC. These suspended computations are reconsidered when the remaining workers perform the completion operation. Thus, an SCC $S$ is completely evaluated when the following two conditions hold:

- There are no unconsumed answers in any consumer node belonging to $S$ or in any consumer node within a suspended SCC in a node belonging to $S$.
- There are no other representations of the leader node $L$ in the computational environment. In other words, $L$ cannot be found in the execution stacks of a different worker, and $L$ cannot be found in the suspended stack segments for another SCC.

Knowing that worker $W$ is at the current leader node $L$ for an SCC $S$, the algorithm for public completion is actually quite straightforward:

- *Atomically* check whether $W$ is the last worker at node $L$, and remember the result as a boolean variable `LastWorkerAtNode`.
- Check if there are unconsumed answers in any consumer node belonging to $S$ or in any consumer node within a suspended SCC in a node belonging to $S$. If so, resume and move to this work.
- If `LastWorkerAtNode` is false, suspend the current SCC and call the *scheduler* to get a new piece of unexploited work.
- Otherwise, if `LastWorkerAtNode` is true, $W$ has completed.

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

The synchronization corresponds to checking beforehand whether $W$ is the last worker, and if so, complete. Note that $W$'s code must take care to check whether $W$ is last before it checks for uncompleted answers, as new answers or nodes might have been generated meanwhile.

A worker $W$ enters in scheduling mode when it runs out of work and only returns to execution mode when a new piece of unexploited work is assigned to it by the scheduler. The scheduler must efficiently distribute the available work for exploitation between workers. OPTYAP has the extra constraint of keeping the correctness of sequential tabling semantics. The OPTYAP scheduler is essentially the YAPOR scheduler (Rocha *et al.* 1999): *when a worker runs out of work it searches for the nearest unexploited alternative in its branch. If there is no such alternative, it selects a busy worker with excess of work load to share work with. If there is no such a worker, the idle worker tries to move to a better position in the search tree.* However, some extensions were introduced in order to preserve the correctness of tabling semantics and ensure that a worker never moves above a leader until it has fully exploited all alternatives. Thus, OPTYAP introduces the constraint that the *computation cannot flow outside the current SCC, and workers cannot be scheduled to execute at nodes older than their current leader node.*

Parallel execution of tabled programs in OPTYAP showed that the system was able to achieve the excellent speedups up to 32 workers for applications with coarse-grained parallelism and quite good speedups for applications with medium parallelism (Rocha *et al.* 2005b).

## 8 Future challenges

Prolog is a well-known language. It is widely used, and is a remarkably powerful tool. The core of Prolog has been very stable throughout the years, both in terms of language design and implementation. Yet, there have been several developments, many within the Logic Programming community, and many more outside. Addressing these developments and the needs of a world very different from when Prolog was created presents both difficulties and opportunities. Next, we discuss some of these issues from our personal perspective.

*Compiler Implementation Technology:* Implementation technology in Prolog needs to be rethought. At the low level, only GNU Prolog currently generates native code (Diaz and Codognet 2001). Just-In-Time technology is a natural match to Prolog and it has shown to work well, but we have just scratched the surface (da Silva and Santos Costa 2007). Progress in compilers, such as `GCC`, may make compilation to `C` affordable again. At a higher level, more compile-time optimization should be done. Determinacy detection is well known (Dawson *et al.* 1995) and should be available. Simple techniques, such as query reordering, can change the program performance hugely for database queries. They should be easily available.

A step further: code expansion for recursive procedures is less of a problem, so why not rethink old ideas such as Krall's VAM (Krall 1996), and Beer's uninitialized

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

variables (Beer 1989; Roy and Despain 1992)? Moreover, years of experience with Ciao Prolog should provide a good basis for rethinking global analysis (Bueno *et al.* 1999).

Last, but not least, Prolog implementation is not just about pure Horn clauses. Challenges such as negation (Sagonas *et al.* 1997) and coinduction (Simon *et al.* 2006) loom large over the future of Logic Programming.

*Language Technology:* At this point in time, there is no dominant language nor framework. But, arguably, some lessons can be taken:

- *Libraries and Data Structures*: Languages need to provide useful and reusable code.
- *Interfacing*: It should be easy to communicate with other languages, and especially with domain-specific languages, such as `SQL` for databases and `R` for statistics.
- *Typing*: It is not clear whether static typing is needed, but it is clear that it is useful, and it is popular in the research community.

Our belief is that progress in this area requires collaboration between different Prolog systems, namely, so that it will be easy to reuse libraries and code. YAP and SWI-Prolog are working together in this direction.

*Logic Programming Technology:* Experience has shown that it is hard to move the results from Logic Programming research to Prolog systems. One illustrative example is XSB Prolog (Sagonas *et al.* 1997): on the one hand, the XSB system has been a vehicle for progress in Logic Programming, supporting the tabling of definite and normal programs. On the other hand, progress in XSB has not been widely adopted. After more than 10 years, even tabling of definite programs is not widely available in other Prolog systems.

The main reason for that is complexity: it is just very hard to implement some of the novel ideas proposed in Logic Programming. Old work suggests that Logic Programming itself may help in this direction (Chen and Warren 1993). Making it easy to change and control Prolog execution in a flexible way is a fundamental challenge for Prolog.

*The WWW:* It has become very important to be able to reason and manipulate data on the world-wide web. Surprisingly, one can see relatively little contribution from the Logic Programming community, although it should be clear that Prolog can have a major role to play, especially related to the semantic web (Wielemaker *et al.* 2008). Initial results offer hope that YAPTAB is competitive with specialized systems in this area (Liang *et al.* 2009).

*Uncertainty:* The last few years have seen much interest in what is often called Statistical Relational Learning (SRL). Several languages designed for this purpose build directly upon Prolog. PRISM (Sato and Kameya 2001) is one of the most popular examples: progress in PRISM has stimulated progress in the underlying Prolog system, B-Prolog (Zhou 2007). Problog is an exciting recent development, and supporting Problog has already lead to progress in YAP (Kimmig *et al.* 2008).

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

Note that even SRL languages that *do not* rely on Prolog offer interesting challenges to the Prolog community. As an interesting example, Markov Logic Networks (MLNs) (Richardson and Domingos 2006) are a popular SRL language that uses bottom-up inference and incremental query evaluation, two techniques that have been well researched in Logic Programming.

## 9 Conclusions and future work

We presented the YAP system, gave the main principles of its implementation, and detailed what we believe are the main contributions in the design of the system, such as engine design, just-in-time indexing, tabling, and parallelism. Arguably, these contributions have made YAP a very competitive system in Prolog applications that require access to large amounts of data, such as learning applications.

Our experience, both as implementers and as users, shows that there are a number of challenges to Prolog. We would like to make "Prolog" faster, more attractive to the Computer Science community and, above all, more useful. To do so, much work has still to be done. Some of the immediate work ahead includes integrating the just-in-time clause compilation framework in the main design of the system, improving performance for attributed variables and constraint systems, improving compatibility with other Prolog systems, and, as always, fixing bugs.

We discussed some of the main challenges that in our opinion face Logic Programming above. YAP has also shown to be an useful platform for work in the languages that combine Prolog and probabilistic reasoning, such as CLP($\mathcal{BN}$) (Santos Costa *et al.* 2008), ProbLog (Kimmig *et al.* 2008), and CPlint (Riguzzi 2007). As argued above, we believe that this is an important research direction for the Logic Programming community, and plan to pursue this work further.

## Acknowledgements

YAP would not exist without the support of the YAP users. We would like to thank them first. The work presented in this paper has been partially supported by project HORUS (PTDC/EIA-EIA/100897/2008), LEAP (PTDC/EIA-CCO/112158/2009), and funds granted to *LIACC* and *CRACS & INESC-Porto LA* through the *Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia* and *Programa POSI*. Last, but not least, we would like to gratefully acknowledge the anonymous referees and the editors of the special number for the major contributions that they have given to this paper.

## References

Aggoun, A., Chan, D., Dufresne, P., Falvey, E., Grant, H., Herold, A., Macartney, G., Meier, M., Miller, D., Mudambi, S., Perez, B., van Rossum, E., Schimpf, J., Tsahageas, P. A. and de Villeneuve, D. H. 1995. *ECL$^i$PS$^e$ 3.5 User Manual*. ECRC.

Ali, K. and Karlsson, R. 1990. Full prolog and scheduling OR-parallelism in muse. *International Journal of Parallel Programming 19*, 6, 445–475.

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

Arnold, D. J., Krauwer, S., Rosner, M., des Tombe, L. and Varile, G. B. 1986. The $< C, A >, T$ framework in Eurotra: A theoretically committed notation for MT. In *Proceedings of the 11th Conference on Computational linguistics*. Association for Computational Linguistics, Morristown, NJ, 297–303.

Beer, J. 1989. *Concepts, Design, and Performance Analysis of a Parallel Prolog Machine*. Lecture Notes in Computer Science, Vol. 404. Springer Verlag, Berlin, DE.

Benton, W. C. and Fischer, C. N. 2007. Interactive, scalable, declarative program analysis: From prototype to implementation. In *Proceedings of the 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, Wroclaw, Poland, 14–16 July 2007, M. Leuschel and A. Podelski, Eds. ACM, New York, NY, 13–24.

Bueno, F., García de la Banda, M. and Hermenegildo, M. 1999. Effectivness of abstract interpretation in automatic parallelization: A case study in logic programming. *ACM Transactions on Programming Languages and Systems 21*, 2, 189–239.

Camacho, R. 1994. Learning stage transition rules with Indlog. In *Proceedings of the 4th International Workshop on Inductive Logic Programming*. GMD-Studien, vol. 237. Gesellschaft für Mathematik und Datenverarbeitung MBH, 273–290.

Carlsson, M. 1987. Freeze, indexing, and other implementation issues in the wam. In *Proceedings of the 4th International Conference on Logic Programming*, J.-L. Lassez, Ed. MIT Press Series in Logic Programming. University of Melbourne, MIT Press, Cambridge, MA, USA, 40–58.

Carlsson, M. 1990. *Design and Implementation of an OR-Parallel Prolog Engine*. PhD Thesis, The Royal Institute of Technology. SICS Dissertation Series 02.

Carlsson, M. and Widen, J. 1988. *SICStus Prolog User's Manual*. Tech. Rep. SICS Research Report R88007B, Swedish Institute of Computer Science, Kista, Sweden.

Castro, L. F. and Costa, V. S. 2001. Understanding memory management in prolog systems. In *Proceedings of the 17th International Conference on Logic Programming, ICLP 2001*. Lecture Notes in Computer Science, vol. 2237. Paphos, Cyprus, 11–26.

Chen, W. and Warren, D. S. 1993. Query evaluation under the well-founded semantics. In *Proceedings of 12th PODS*. 168–179.

Chico, P., Carro, M., Hermenegildo, M. V., Silva, C. and Rocha, R. 2008. An improved continuation call-based implementation of tabling. In *Proceedings of the International Symposium on Practical Aspects of Declarative Languages*. LNCS, vol. 4902. Springer-Verlag, Berlin, DE, 197–213.

Colmerauer, A. 1993. The birth of prolog. In *The Second ACM-SIGPLAN History of Programming Languages Conference*. ACM, New York, NY, USA, 37–52.

Costa, V. S. 1988. Implementação de prolog. Provas de aptidão pedagógica e capacidade científica, Universidade do Porto.

Costa, V. S. 1999. Optimising bytecode emulation for prolog. In *Proceedings of PPDP 1999*. LNCS, vol. 1702. Springer-Verlag, Berlin, DE, 261–267.

Costa, V. S. 2007. Prolog performance on larger datasets. In *Proceedings of the 9th International Symposium on Practical Aspects of Declarative Languages, PADL 2007*, Nice, France, 14–15 January 2007, M. Hanus, Ed. Lecture Notes in Computer Science, vol. 4354. Springer, Berlin, DE, 185–199.

Costa, V. S. 2008. The life of a logic programming system. In *Proceedings of the 24th International Conference on Logic Programming, ICLP 2008*, Udine, Italy, 9–13 December 2008, M. G. de la Banda and E. Pontelli, Eds. Lecture Notes in Computer Science, vol. 5366. Springer, Berlin, DE, 1–6.

Costa, V. S. 2009. On just in time indexing of dynamic predicates in prolog. In *Progress in Artificial Intelligence, 14th Portuguese Conference on Artificial Intelligence, EPIA 2009*,

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

L. S. Lopes, N. Lau, P. Mariano and L. Rocha, Eds. Lecture Notes in Computer Science. Springer, Berlin, DE, 126–137.

Costa, V. S., de Castro Dutra, I. and Rocha, R. 2010. Threads and or-parallelism unified. *TPLP 10*, 4–6, 417–432.

Costa, V. S., Page, D. and Cussens, J. 2008. Clp($n$): Constraint logic programming for probabilistic knowledge. In *Proceedings of Probabilistic Inductive Logic Programming—Theory and Applications*, L. D. Raedt, P. Frasconi, K. Kersting and S. Muggleton, Eds. Lecture Notes in Computer Science, vol. 4911. Springer, Berlin, DE, 156–188.

Costa, V. S., Sagonas, K. and Lopes, R. 2007. Demand-driven indexing of prolog clauses. In *Proceedings of the 23rd International Conference on Logic Programming*, V. Dahl and I. Niemelä, Eds. Lecture Notes in Computer Science, vol. 4670. Springer, Berlin, DE, 305–409.

da Silva, A. F. and Costa, V. S. 2007. Design, implementation, and evaluation of a dynamic compilation framework for the yap system. In *Proceedings of the 23rd International Conference on Logic Programming, ICLP 2007,* 8–13 September 2007, Lecture Notes in Computer Science, vol. 4670. Springer, Berlin, DE, 410–424.

Damas, L. and Milner, R. 1982. Principal type-schemes for functional programs. In *POPL*. 207–212.

Davis, J., Dutra, I., Page, D. and Costa, V. S. 2005. Establishing identity equivalence in multi-relational domains. In *Proceedings of the 2005 International Conference on Intelligence Analysis*.

Dawson, S., Ramakrishnan, C. R., Ramakrishnan, I. V., Sagonas, K. F., Skiena, S., Swift, T. and Warren, D. S. 1995. Unification factoring for efficient execution of logic programs. In *POPL 1995*. ACM Press, New York, 247–258.

Demoen, B., Janssens, G. and Vandecasteele, H. 2000. Compiling large disjunctions. In *Proceedings of the Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages, London*.

Demoen, B., Mariën, A. and Callebaut, A. 1989. Indexing in prolog. In *Proceedings of the North American Conference on Logic Programming*, E. L. Lusk and R. A. Overbeek, Eds. 1001–1012.

Demoen, B. and Nguyen, P.-L. 2000. So many WAM variations, so little time. In *Proceedings of the International Conference on Computational Logic—CL 2000*. LNAI, vol. 1861. Springer-Verlag, Berlin, DE, 1240–1254.

Devitt, S., Roo, J. D. and Chen, H. 2005. Desirable features of rule based systems for medical knowledge. In *Proceedings of the W3C Workshop on Rule Languages for Interoperability,* Washington, DC, 27–28 April 2005, W3C.

Diaz, D. and Codognet, P. 2001. Design and implementation of the GNU prolog system. *Journal of Functional and Logic Programming 2001,* 2001.

Filgueiras, M. 1984. A prolog interpreter working with infinite terms. In *Implementations of Prolog*. Campbell, Halsted Press, Chichester, West Sussex, 250–258.

Fonseca, N. A., Costa, V. S., Rocha, R., Camacho, R. and Silva, F. M. A. 2009. Improving the efficiency of inductive logic programming systems. *Software, Practice and Experience 39*, 2, 189–219.

Granlund, T. 2004. GNU multiple precision arithmetic library 4.1.4.

Hermenegildo, M. V., Bueno, F., Carro, M., López, P., Morales, J. F. and Puebla, G. 2008. An overview of the ciao multiparadigm language and program development environment and its design philosophy. In *Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, P. Degano, R. D. Nicola and J. Meseguer, Eds. Lecture Notes in Computer Science, vol. 5065. Springer, Berlin, DE, 209–237.

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

Kimmig, A., Costa, V. S., Rocha, R., Demoen, B. and Raedt, L. D. 2008. On the efficient execution of problog programs. In *Proceedings of the 24th International Conference on Logic Programming, ICLP 2008,* Udine, Italy, 9–13 December 2008, M. G. de la Banda and E. Pontelli, Eds. Lecture Notes in Computer Science, vol. 5366. Springer, Berlin, DE, 175–189.

Krall, A. 1996. The vienna abstract machine. *Journal of Logic Programming 29,* 85–106.

Lea, D. 1996. *A Memory Allocator*. URL: http://gee.cs.oswego.edu/dl/html/malloc.html

Liang, S., Fodor, P., Wan, H. and Kifer, M. 2009. OpenRuleBench: An analysis of the performance of rule engines. *Proceedings of the 18th International Conference on World Wide Web*. 601–610.

Mariën, A. 1993. Improving the compilation of prolog in the framework of the warren abstract machine. PhD Thesis, Katholiek Universiteit Leuven, Leuven, BE.

Mariën, A. and Demoen, B. 1989. On the management of choicepoint and environment frames in the WAM. In *Proceedings of the North American Conference on Logic Programming*, Cleveland, OH, E. L. Lusk and R. A. Overbeek, Eds. 1030–1050.

Mariën, A. and Demoen, B. 1991. A new scheme for unification in WAM. In *Proceedings of the 1991 International Symposium on Logic Programming*, V. A. Saraswat and K. Ueda, Eds. MIT Press, San Diego, 257–271.

Meier, M. 1990. Compilation of compound terms in prolog. In *Proceedings of the 1990 North American Conference on Logic Programming*, S. K. Debray and M. V. Hermenegildo, Eds. MIT Press, Austin, TX, 63–79.

Morales, J. F., Carro, M. and Hermenegildo, M. V. 2008. Comparing tag scheme variations using an abstract machine generator. In *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, Valencia, Spain, 15–17 July 2008, S. Antoy and E. Albert, Eds. ACM, New York, NY, USA, 32–43.

Mungall, C. 2009. Experiences using logic programming in bioinformatics. In *Proceedings of the 25th International Conference on Logic Programming, ICLP 2009,* Pasadena, CA, 14–17 July 2009, P. M. Hill and D. S. Warren, Eds. Lecture Notes in Computer Science, vol. 5649. Springer, Berlin, DE, 1–21.

Näsén, H., Carlsson, M. and Sagonas, K. F. 2001. Instruction merging and specialization in the sicstus prolog virtual machine. In *Proceedings of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming,* Florence, Italy, 5–7 September 2001, ACM, New York, NY, 49–60.

Nugues, P. M. 2006. *An Introduction to Language Processing with Perl and Prolog: An Outline of Theories, Implementation, and Application with Special Consideration of English, French, and German (Cognitive Technologies)*. Springer-Verlag New York, Secaucus, NJ.

Page, D. and Srinivasan, A. 2003. Ilp: A short look back and a longer look forward. *Journal of Machine Learning Research 4*, 415–430.

Pereira, F. 1987. *C-Prolog 1.5 User Manual*. SRI International, Menlo Park, CA.

Quintus 1986. *Quintus Prolog User's Guide and Reference Manual—Version 6.*

Ramakrishnan, I. V., Rao, P., Sagonas, K., Swift, T. and Warren, D. S. 1999. Efficient access mechanisms for tabled logic programs. *Journal of Logic Programming 38*, 1, 31–54.

Richardson, M. and Domingos, P. 2006. Markov logic networks. *Machine Learning 62*, 107–136.

Riguzzi, F. 2007. A top down interpreter for lpad and cp-logic. In *Proceedings of the 10th Congress of the Italian Association for Artificial Intelligence on Artificial Intelligence and Human-Oriented Computing, AI*IA 2007*, Rome, Italy, 10–13 September 2007, R. Basili and M. T. Pazienza, Eds. Lecture Notes in Computer Science, vol. 4733. Springer, Berlin, DE, 109–120.

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

Rocha, R. 2006. Handling incomplete and complete tables in tabled logic programs. In *Proceedings of the International Conference on Logic Programming*. LNCS, vol. 4079. Springer-Verlag, Berlin, DE, 427–428.

Rocha, R. 2007. On improving the efficiency and robustness of table storage mechanisms for tabled evaluation. In *Proceedings of the International Symposium on Practical Aspects of Declarative Languages*. LNCS, vol. 4354. Springer-Verlag, Berlin, DE, 155–169.

Rocha, R., Silva, F. and Costa, V. S. 1999. YapOr: An Or-parallel prolog system based on environment copying. In *Portuguese Conference on Artificial Intelligence*. LNAI, vol. 1695. Springer-Verlag, Berlin, DE, 178–192.

Rocha, R., Silva, F. and Costa, V. S. 2000. YapTab: A tabling engine designed to support parallelism. In *Conference on Tabulation in Parsing and Deduction*. 77–87.

Rocha, R., Silva, F. and Costa, V. S. 2002. Achieving scalability in parallel tabled logic programs. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPPDPS)*, Fort Lauderdale, FL, 51.

Rocha, R., Silva, F. and Costa, V. S. 2005a. Dynamic mixed-strategy evaluation of tabled logic programs. In *Proceedings of the International Conference on Logic Programming*. LNCS, vol. 3668. Springer-Verlag, Berlin, DE, 250–264.

Rocha, R., Silva, F. and Costa, V. S. 2005b. On applying Or-parallelism and tabling to logic programs. *Theory and Practice of Logic Programming Systems 5*, 1/2, 161–205.

Roy, P. V. and Despain, A. M. 1992. High-performance logic programming with the aquarius prolog compiler. *IEEE Computer 25*, 1, 54–68.

Sagonas, K. and Swift, T. 1998. An abstract machine for tabled execution of fixed-order stratified Logic Programs. *ACM Transactions on Programming Languages and Systems 20*, 3, 586–634.

Sagonas, K. F., Swift, T., Warren, D. S., Freire, J. and Rao, P. 1997. The XSB programmer's manual. Tech. Rrep., State University of New York at Stony Brook. URL: http://xsb.sourceforge.net/

Sato, T. and Kameya, Y. 2001. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research 15*, 391–454.

Schimpf, J. 1990. Garbage collection for prolog based on twin cells. In *2nd NACLP Workshop on Logic Programming Architectures and Implementations*. MIT Press, Cambridge, MA, USA.

Schrijvers, T. 2008. Constraint handling rules. In *Proceedings of the 24th International Conference on Logic Programming, ICLP 2008,* Udine, Italy, 9–13 December 2008, M. G. de la Banda and E. Pontelli, Eds. Lecture Notes in Computer Science, vol. 5366. Springer, Berlin, DE, 9–10.

Simon, L., Mallya, A., Bansal, A. and Gupta, G. 2006. Coinductive logic programming. In *Proceedings of the 22nd International Conference on Logic Programming, ICLP 2006*, Seattle, WA, 17–20 August 2006, S. Etalle and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 4079. Springer, Berlin, DE, 330–345.

Somogyi, Z. and Sagonas, K. 2006. Tabling in mercury: Design and implementation. In *Proceedings of the International Symposium on Practical Aspects of Declarative Languages*. LNCS, vol. 3819. Springer-Verlag, Berlin, DE, 150–167.

Srinivasan, A. 2001. *The Aleph Manual*.

Tarau, P. and Neumerkel, U. 1994. A novel term compression scheme and data representation in the binwam. In *Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming, PLILP 1994*, Madrid, Spain, 14–16 September 1994, M. V. Hermenegildo and J. Penjam, Eds. Lecture Notes in Computer Science, vol. 844. Springer, Berlin, DE, 73–87.

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press

Tarjan, R. E. 1972. Depth-first search and linear graph algorithms. *SIAM Journal on Computing 1*, 2, 146–160.

Tronçon, R., Janssens, G., Demoen, B. and Vandecasteele, H. 2007. Fast frequent querying with lazy control flow compilation. *TPLP 7*, 4, 481–498.

Van Roy, P. 1990. Can logic programming execute as fast as imperative programming? PhD Thesis, University of California, Berkeley, CA.

Vandeginste, R. and Demoen, B. 2007. Incremental copying garbage collection for WAM-based Prolog systems. *TPLP 7*, 5, 505–536.

Vaz, D., Costa, V. S. and Ferreira, M. 2009. User defined indexing. In *Proceedings of the 25th International Conference on Logic Programming, ICLP 2009,* Pasadena, CA, 14–17 July 2009, P. M. Hill and D. S. Warren, Eds. Lecture Notes in Computer Science, vol. 5649. Springer, Berlin, DE, 372–386.

Warren, D. H. D. 1983. An abstract prolog instruction set. Technical Note 309, SRI International, Menlo Park, CA, USA.

Wielemaker, J. 2003. Native preemptive threads in SWI-prolog. In *Proceedings of the 19th International Conference on Logic Programming, ICLP 2003*, Mumbai, India, 9–13 December 2003, C. Palamidessi, Ed. Lecture Notes in Computer Science, vol. 2916. Springer, Berlin, DE, 331–345.

Wielemaker, J. 2010. *SWI-Prolog 5.9.9: Reference Manual*. Department of Computer Science, VU University Amsterdam, Amsterdam, The Netherlands.

Wielemaker, J., Hildebrand, M., van Ossenbruggen, J. and Schreiber, G. 2008. Thesaurus-based search in large heterogeneous collections. In *Proceedings of the 7th International Semantic Web Conference on the Semantic Web, ISWC 2008,* Karlsruhe, Germany, 26–30 October 2008, LNCS, vol. 5318. Springer, Berlin, DE, 695–708.

Zhou, N.-F. 2007. A register-free abstract prolog machine with jumbo instructions. In *Proceedings of the 23rd International Conference on Logic Programming, ICLP 2007,* Porto, Portugal, 8–13 September 2007, Lecture Notes in Computer Science, vol. 4670. Springer, Berlin, DE, 455–457.

Zhou, N.-F., Takagi, T. and Kazuo, U. 1990. A matching tree oriented abstract machine for prolog. In *Proceedings of the 7th International Conference on Logic Programming*, D. H. D. Warren and P. Szeredi, Eds. MIT Press, Cambridge, MA, USA, 158–173.

https://doi.org/10.1017/S1471068411000512 Published online by Cambridge University Press