

The XSB Logic Programming System

The XSB Research Group

The goal of the XSB project, which continues under active development at SUNY Stony Brook and Universidade Nova de Lisboa, is to extend Prolog with features for deductive databases and non-monotonic reasoning. The resulting system has proven useful in a wide variety of research areas, and has been heavily used by a number of commercial companies.

At the core of XSB is an industrial-strength, multi-threaded Prolog system that is largely ISO-compliant, executes in both 32-bit and 64-bit modes, and has full support for various forms of tabling and constraint-handling rules at the engine level. Numerous libraries and packages have been written for XSB that allow sophisticated interfaces to Java, C, and to various database systems. Other packages allow GUI development from within XSB, provide for ontology management, and support other functionality. We discuss some of the more significant features in the following.

- *Tabling* The most distinctive aspect of XSB is its implementation of tabled evaluation. Tabling is a simple idea with far-reaching implications. At a conceptual level, during computation of a goal to a logic program, each subgoal S is registered in a table the first time it is called, and unique answers to S are added to the table as they are derived. When subsequent calls are made to S , the evaluation ensures that answers to S are retrieved from the table rather than being re-derived using program clauses. Even from this simple description, an advantage of tabling can be seen – it ensures termination for various classes of programs. Consider the case of positive datalog programs, i.e. Horn Clause programs with terms consisting of only variables and constants from a finite alphabet. Such a program contains only finitely many atomic formulas. Each of these atoms can be added at most once to the table as a subgoal, and each such subgoal can have at most finitely many answers, leading to a finite computation.

However, tabling can be used to far greater effect than simply ensuring termination for some Horn clause programs. One powerful feature of tabling is its ability to maintain other global elements of a computation in the “table,” such as information about whether one subgoal depends on another, and whether the dependency is through negation. By maintaining this global information, tabling can be used to evaluate normal logic programs under the Well-Founded Semantics (WFS). The essential idea is that global information about dependencies is used to determine the truth value of

literals that do not have a derivation. If such literals are involved in a cyclic dependency through negation, they are undefined under WFS; if not, the literals belong to an unfounded set and are false in WFS. In fact, it can be shown that tabling allows non-floundering datalog programs with negation to terminate with quadratic data complexity under WFS. Of course not every goal in an evaluation must be tabled: users can choose whether individual predicates use tabled evaluation, or can allow the system to decide which predicates in a given module should be tabled.

- *Call-Variance and Call-Subsumption* By default, a *call-variant* evaluation is performed – in other words, a goal G uses a table if the table contains a goal G' that is a variant of G . Call-variance is a useful form of tabling when it is desirable to preserve certain meta-logical properties of an evaluation. As one example, when a meta-interpreter is tabled using call-variance, it still functions as a meta-interpreter, but will have the semantic, termination, and complexity properties of the tabled program. Call-variant tabling has proven useful in applications including program analysis, ontology management, diagnosis, model checking, machine learning, and natural language analysis. Applications in many of these areas make heavy use of well-founded tabled negation in addition to tabling Horn clauses.

Alternatively, a predicate can be declared to use *call-subsumption* where a goal G uses a table if the table contains a goal G' that *subsumes* G . This seemingly minor change gives rise to very different evaluation properties. If a goal is called with an uninstantiated goal G_{uninst} , whenever a subgoal is encountered with the same predicate symbol as G_{uninst} the table for G_{uninst} will be used, and so the evaluation can be made to behave as a bottom-up computation of a least fixed point. Call subsumption is most useful for semantics that are modeled as simple fixed points, such as inferencers for RDF and other semantic web tools.

- *Tabled Aggregation* Tabled aggregation does not preserve all answers for a goal, but only those that are deemed “best” according to a given measure. If this measure is a lattice, the table may maintain only the join of all answers to a goal; if the measure is a simple partial order, the table may maintain only the maximal answers to a goal. The ability to maintain the join of answers in a lattice has proven useful for reactive agents and semantic classifiers through implementations of para-consistent, fuzzy and possibilistic logics.
- *Incremental Table Maintenance* XSB has recently been extended with a capability for incremental table maintenance (aka view maintenance or truth maintenance.) When a user declares a table to be incrementally maintained, the table will be kept up-to-date when changes are made to dynamic predicates on which the table depends. This facility allows XSB to easily support the “model-view-controller” architecture of interactive systems: dynamic predicates store facts constituting the model; tables maintain the views defined in terms of these facts that are presented to the user; and the controller takes user input, modifies the dynamic predicates appropriately, and efficiently updates the tabled views through incremental table

maintenance.

- *Support for Constraints* Like many other Prologs, XSB offers support for Constraint Logic Programming, doing so through an implementation of *Constraint Handling Rules (CHR)*. However XSB also provides the ability to combine tabling and constraints in the same evaluation. When a tabled goal G is encountered with a set of constraints C on its variables, a check is made to determine whether the table contains a variant goal with the same constraints: if so, the table is used. When an answer is resolved against a goal, unifications in its resolution may spark constraint evaluation just as in traditional CLP evaluations. The combination of tabling and constraints has proven useful for applications such as verification of security protocols and of real-time systems.
- *Multi-threading* XSB supports a draft ISO-standard for multi-threaded Prologs based on the semantics of Unix Pthreads and Windows threads. XSB's multi-threaded implementation allows numerous threads to be created quickly and to execute with high concurrency. Threads can be created using a command such as

```
?- thread_create(Goal,Child_id).
```

which creates a thread with id `Child_id` to execute `Goal`. Within the calling thread, `thread_create/2` returns immediately, and `Child_id` executes `Goal` asynchronously, without directly sharing bindings with the calling thread. Each thread has all the functionality of a sequential engine. By default, tables and dynamic code are thread-private, so each thread can concurrently execute tabling under WFS (perhaps with tabled constraints), can assert and retract code to a private store (including asserting tabled code), and so on. This design, which requires minimal synchronization, supports scalability of evaluations when dozens or more threads are used, and these evaluations can exploit the newer multi-core, 64-bit architectures.

Threads can communicate through message queues, through shared dynamic predicates, or through shared tables. Message queues are either private to a thread (so that the queue can be read only by that thread) or public. A thread accessing a message queue will, if no message is found, suspend until a message is added. Information in thread-shared dynamic code or in a thread-shared table that is completed (i.e. one for which all available answers have been derived) can be read by any thread. Shared tables computed by different threads can depend on one another; by default thread-shared tables in XSB use an optimistic concurrency control mechanism that prevents two threads from creating the same table at the same time.

- *Dynamic Code* One goal of the XSB system is to support deductive database applications, which critically depend on database update, so much work has gone into XSB's implementation of dynamic code and facts. The first type of dynamic code is standard in Prolog: clauses may be added by `assert/1` and related predicates, and removed by `retract/1` and related predicates. XSB differs from other Prologs, however, in the

varieties of indexing it supports. The indexing of a dynamic predicate p/n is specified by a declaration $\text{index}/2$. Clauses may be indexed on any argument or on alternate arguments. E.g., $\text{- index}(p/6, [3,2])$ means to use an index on the outer functor symbol of argument 3 when a call to $p/6$ has its third argument bound, and use an index on argument 2 when argument 3 is a variable, but argument 2 is bound. Dynamic clauses may also be indexed on combinations of arguments if a single argument does not provide enough discrimination for good indexing – thus $\text{- index}(p/6, [3+2])$ forms an index from the outer functor symbol of argument 3 together with the outer functor symbol of argument 2. Finally *star-indexing* allows discrimination on information that may be in the first 5 symbols of a term, so that dynamic code containing lists or other complex structures may be effectively indexed. Standard dynamic code is thread-private by default, but can be declared thread-shared at the predicate level. Asserting and retracting clauses in XSB is fast: millions of clauses can be read and asserted in a few seconds.

Trie-indexed dynamic code offers an alternate mechanism for storing and maintaining large numbers of facts. Trie-indexed code compiles facts into trie-instructions similar to those used for XSB’s tables. For instance set of facts $\{ \text{rt}(a, f(a, b), a), \text{rt}(a, f(a, X), Y), \text{rt}(b, V, d) \}$ would be stored in a trie as shown in Figure 1, where each node corresponds to an instruction in XSB’s virtual machine.

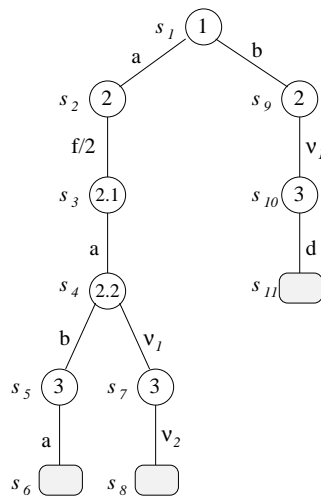


Figure 1: Terms Stored as a Trie

Using a trie for indexing has the advantage that discrimination can be made on a position anywhere in a fact, and asserts and retracts are 2-3x faster than with standard dynamic code. In addition, in trie-dynamic code, there is no distinction between the index and the code itself, so for many sets of facts trie indexing can use much less space than standard dynamic code. However, trie indexing comes with tradeoffs: only facts can be made trie-dynamic, and unlike standard dynamic code, no ordering is preserved among the facts and duplicate facts are not supported.

Based on the engine described above, a number of packages and libraries have been written for XSB (some by groups independent of the XSB Group).

- *InterProlog and XJ* InterProlog (<http://www.declarativa.com/interprolog>) is a library that allows Java programs to communicate with XSB, supporting the inter-conversion of objects between Java and Prolog datatypes. InterProlog supports either socket or JNI communication between Java and XSB. InterProlog has been used to implement XJ (<http://www.xsb.com/xj.aspx>), a subsystem that allows XSB programmers to create GUIs using the Java Swing Library. The XSB programmer sends Prolog terms that represent GUI objects to XJ, which creates the desired Swing object and manages callbacks to XSB to handle user interactions. Complex and sophisticated interfaces can be created from XSB without any Java programming.
- *XASP* allows users to integrate Prolog and answer set programming (ASP). An answer set program can be passed to Smodels (<http://www.tcs.hut.fi/Software/smodels>) directly from a *residual* program, consisting of atoms whose truth could not be determined during a well-founded evaluation and their inter-dependencies. Alternately, an ASP program can be explicitly constructed and sent to Smodels. Furthermore, each XSB thread has the ability to invoke its own private copy of Smodels within the XSB process, leading to the ability to efficiently compute disjunctive programs, preferred stable models and other extensions of stable model computation. The XASP package has proven useful for implementations of collaborative agents and logics of change.
- The *Coherent Description Framework* (CDF) is a library for maintaining ontology-style information in XSB. Users can define an inheritance hierarchy containing classes, along with objects that are members of those classes. Objects in each class may have mandatory or optional relationships to objects in other classes: these relationships may be defined at the level of the objects themselves or of the classes that contain them. Definitions of classes, objects, and their relationships may be either extensional (consisting of dynamic facts) or intensional (consisting of dynamic rules). A variety of routines are provided for quickly loading and updating ontologies with millions of elements: alternatively the ontology may be stored in a database and lazily accessed. When queries are made to a CDF ontology, answers are correct both according to WFS and to a description logic-like semantics. CDF has been used extensively at XSB, Inc. (<http://www.xsb.com>) for semantic-web applications involving knowledge extraction and classification.
- For those who want a programming environment that combines logic and objects more thoroughly than CDF, *Flora* (<http://flora.sourceforge.net>) amalgamates F-logic, Transaction Logic, and HiLog. Flora's syntax differs from Prolog's. The term:
`john[spouse->mary, child->>bob,bill]`

indicates that the object `john` has a unique `spouse`-attribute of `mary` and `child`-attributes `bob` and `bill`. Such an F-logic term is analogous to a Prolog term – it may

appear as a fact, as the head of a rule, or in a literal in the body of a rule. Flora is implemented by an optimizing compiler whose target code is a normal program executable by XSB. This compiler makes heavy use of XSB's indexing mechanisms and negation. However, Flora also has its own sophisticated user environment with a debugger and module system, as well as support for meta-programming of HiLog programs. Flora has been used for numerous research and commercial projects involving the semantic web, network diagnosis and other application areas.

XSB is available from <http://xsb.sourceforge.net> under the GNU Library General Public License, so that it can be freely used for research and commercial applications. Papers about XSB, its implementation and applications can be found on the home pages of its contributing members.