Knowledge Representation in XSB, Flora and Silk

Terrance Swift

May 4, 2012
Part 2: System KR Features

1. XSB and Flora
2. Frame Logic
3. HiLog
4. Transaction Logic
5. Defeasible Reasoning
6. Understanding Computations
7. Discussion
XSB combines tabled and non-tabled predicates.

- A predicate $p/n$ is non-tabled, unless it is declared to be tabled by a declaration such as `:- table p/n`.
- For non-tabled predicates, the operator for negation is `\+`, while for tabled predicates the operator for negation is `\not/1` (or `\sknot/1` for non-ground negation).
  - `\+` provides Prolog semantics, while `\not/1` provides well-founded semantics.
- In general a Prolog predicate may succeed, fail or throw an exception.
  - Question: when do KR systems actually need exceptions?
- In XSB, a predicate that succeeds may be true (unconditional) or undefined (conditional).
Flora-2

- \texttt{FLORA-2} is an XSB-based implementation of F-logic, transaction logic, HiLog, and much more. \texttt{FLORA-2} has its own command-line interpreter, debugger, and other basic elements of a programming environment.

- \texttt{FLORA-2} predicates may call XSB, and XSB predicates may call \texttt{FLORA-2}. Although \texttt{FLORA-2} terms are represented in Prolog, some translation is usually necessary.

- A predicate \texttt{p/n} is tabled, unless its name begins with the \texttt{%} symbol. Below, \texttt{tc/2} is tabled, but \texttt{edge/2} is not.

  \begin{verbatim}
  tc(X, Y) :- %edge(?X,?Y).
  tc(X, Y) :- tc(?X,?Z).%edge(?X,?Y)
  \end{verbatim}

- Note that variables begin with \texttt{?} rather than with a capital letter as in Prolog.

- The sourceforge version is very old and does not contain many of the features discussed here.
FLORA-2 supports the Prolog-style syntax of using atomic predicates, but it also supports a frame syntax.

F-logic [KLPW95] was a formalism designed to intermix logic and object-orientation.

Basic frame syntax is supported in RIF-BLD (www.w3.org/TR/rif-bld) and RIF-PLD (www.w3.org/TR/rif-prd).

An example is presented on the next slide.
Schema:

- `paper[authors *=> person, title *=> string]`
- `journal_p :: paper[in_vol *=> volume]`
- `conf_p :: paper[at_conf *=> conf_proc]`
- `journal_vol[of *=> journal, volume *=> integer, number *=> integer, year *=> integer]`
- `journal[name *=> string, publisher *=> string, editors *=> person]`
- `conf_proc[of_conf *=> conf_series, year *=> integer, editors *=> person]`
- `conf_series[name *=> string]`
- `publisher[name *=> string]`
- `person[name *=> string, affil(integer) *=> institution]`
- `institution[name *=> string, address *=> string]`

Objects:

- `o_j1 : journal_p [title -> 'Records, Relations, Sets, Entities, and Things', authors -> {o_mes}, in_vol -> o_i11]`
- `o_di : conf_p [title -> 'DIAM II and Levels of Abstraction', authors -> {o_mes, o_eba}, at_conf -> o_v76]`
- `o_i11 : journal_vol[of -> o_is, number -> 1, volume -> 1, year -> 1975]`
- `o_is : journal[name -> 'Information Systems', editors -> {o_mj}]`
- `o_v76 : conf_proc[of -> vldb, year -> 1976, editors -> {o pcl, o ejn}]`
- `o_vldb : conf_series[name -> 'Very Large Databases']`
- `o_mes : person[name -> 'Michael E. Senko']`
- `o_mj : person[name -> 'Matthias Jarke', affil(1976) -> o_rwt]`
- `o_rwt : institution[name -> 'RWTH_Aachen']`
Classes in Flora-2

- paper, person are classes
- :: is the subclass relation, e.g., journal :: paper
- *=> is an inheritable class property, e.g.,
  
  paper[authors *=> person, title *=> string].

  You can think of this as a type declaration.

- => is a non-inheritable class relation. This is used to represent, say, the average value of a set.

- Multiple atomic statements can be gathered together in a molecule:
  
  journal[name*=>string, publisher*=>string, editors*=>person].

- In addition, cardinality constraints can be placed on the number of values for a method. The syntax is:
  
  C1[Meth{LowerBound:UpperBound}=>C12]
  C1[Meth{LowerBound:UpperBound}*=>C12]

  although these constraints must be explicitly checked.
Objects

- \texttt{o\_mes:person[name -> 'Michael E. Senko']} means \texttt{o\_mes} is a person and the objects name is 'Michael E. Senko'
  - : is the class membership relation
  - If \texttt{o\_mes} was known to have a unique name, we could identify the object with the name: 'Michael E. Senko' : person, which might be more clear in some contexts.

- In the \textit{molecule}
  \[
  \texttt{o\_di:conf\_p[title -> 'DIAM II and Levels of Abstraction',}
  \texttt{authors -> \{o\_mes, o\_eba\}, at\_conf -> o\_v76}].
  \]

the symbol \(\rightarrow\) is an “attribute” operator. A query such as

\[
?\leftarrow o\_di[\text{author} \rightarrow o\_mes]
\]

would succeed as \texttt{o\_mes} \(\in\) \texttt{\{o\_mes, o\_eba\}}
In addition to types, default values may also be inherited.

British[nativeLanguage *--> English].

such a value can be overridden by defining a value for a subclass or object:

Joao:British[nativeLanguage --> Portuguese].

Note that since Joao ia an object, there is no need to use inheritance.

Note that O[A->V] or O[A*->V] means that there is a single value V as the A attribute for O (and similarly for -\_).

Multi-valued attributes can be defined via the operators ->-> and *-->--.

Thus for *-->, inheritance is non-monotonic; for *-->-> inheritance is monotonic.
Non-monotonic inheritance leads to the possibility of contradiction.

Consider the so-called “Nixon Diamond” example

\[
\begin{align*}
nixon &: \text{republican}. \\
nixon &: \text{quaker}. \\
\text{republican}[\text{policy } \mapsto \text{ nonpacifist}]. \\
\text{quaker}[\text{policy } \mapsto \text{ pacifist}].
\end{align*}
\]

If no defeasibility theory is used, the policy of \texttt{nixon} would be undefined: nonpacifist if not pacifist and pacifist if not non-pacifist. However the defeasibility theory reduces the occurrences of undefined answers.
HiLog

- HiLog [CKW93] is a framework that provides logic programs with a second-order syntax, but retaining a first-order semantics: the functor symbol of a term or predicate may be a variable, or some other term.

- HiLog allows for traditional higher-order programming. An XSB example:

  closure(R)(X,Y) :- R(X,Y).
  closure(R)(X,Y) :- closure(R)(X,Z),R(Z,Y).

- HiLog also allows easier querying over Prolog-style structures. A FLORA-2 example:

  tvaSentence(?P(?ARG1, ?NEW2, ?ARG3)) :-
    imp#tvaPredicate(?P)
    and assertedSentence(transitiveViaArg(?P, ?LINK, (2)))
    and removalSentence(?P(?ARG1, ?ARG2, ?ARG3))
    and ?ARG2 != ?NEW2
    and removalSentence(?LINK(?ARG2, ?NEW2)).
HiLog thus provides a semantics call/n, functor/3 and arg/3, along with an alternate execution mechanism.

The basic idea is to compile a $n$-ary term $f(\vec{t})$ into a $n+1$-ary term of the form $apply(f,\vec{t})$.

XSB supports HiLog (cf. the closure example), where each HiLog term must be explicitly declared. But HiLog isn’t integrated with the XSB module system.
HiLog in Flora

- \texttt{FLORA-2} supports HiLog more thoroughly, where each user predicate is (generally) assumed to be HiLog. This means that when compiled to XSB, a \texttt{FLORA-2} program is one big predicate.

- Arguably, the frame syntax of \texttt{FLORA-2} makes HiLog more natural. In querying a term of the form \texttt{object[attribute --> value]} it is natural to have \texttt{object}, \texttt{attribute} and/or \texttt{value} as a variable.

- The use of frame-syntax along with the use of HiLog and the module system of \texttt{FLORA-2} lead to the need for various reification and meta-unification operators, particularly when meta-programming.

- In addition, the use of HiLog leads to the need for more tabling, as it becomes more difficult to determine when a predicate may be safely executed without tabling.
Recall that Prolog uses predicates such as assert/1, retract/1 and retractall/1 to updates is knowledge base.

While a “pure” XSB program has the WFS semantics, the update operators fall outside of WFS.

Transaction logic (T-logic) [BK94] is a formalism to capture the semantics of updates in logic programming.
A serial Horn rule has the form:

\[ h :- b_1 \otimes b_2 \otimes \ldots \otimes b_n \]

where \( \otimes \) is the sequential conjunction operator, \( h \) is an atom and \( b_i \) are T-literals.

T-literals are (program) atoms, negated program or the update commands \textit{insert}(A) or \textit{delete}(A) for some atom \( A \).
Transaction Logic Semantics

- If $b_i$ and $b_j$ are atoms, $\otimes \equiv \land$ and so is commutative.
- Otherwise, let $b_i$ is a update command and $b_j$ a literal, and suppose we have a program state $S_i$. Then $b_i \otimes b_j$ means to query $b_j$ in $S_j$, the state where $S_i$ is modified by the update action $b_i$.
- So in general $b_1 \otimes b_2 \otimes \ldots \otimes b_n$ is associated with a sequence of program states e.g., $S_1, S_2, \ldots, S_n$. Note that if $b_j$ is not an update command, $S_j = S_{j-1}$
- $b_1 \otimes b_2 \otimes \ldots \otimes b_n$ is true if it is satisfied in the sequence $S_1, S_2, \ldots, S_n$
- I’m abusing a program state, identifying it with the program (for $\vdash$) and the unique WFM of the program (for $|=\)$. 

Terrance Swift () Knowledge Representation May 4, 2012 16 / 39
Transaction Logic Example

\[ \text{move}(X,Y) \quad \text{:-} \quad \text{on}(X,Z) \otimes \text{clear}(X) \otimes \text{clear}(Y) \]
\[ \otimes \text{tnot tooHeavy}(X) \]
\[ \otimes \text{delete(on}(X,Z)) \]
\[ \otimes \text{insert(on}(X, Y)) \]
\[ \otimes \text{delete(clear}(Y)). \]

\[ \text{tooHeavy}(X) \quad \text{:-} \quad \text{weight}(X,W) \otimes \text{limit}(L) \otimes W < L. \]

?- move(blk1, blk15) \otimes \text{move(}\text{SomeBlk, blk1}). \]
Backtrackable Updates in Flora and XSB

- XSB and FLORA-2 both partially support T-logic.
- The first aspect to support is backtrackable updates for facts. In XSB this is done using the storage library.
- It standard in FLORA-2 via commands such as \texttt{t_insert/1} and \texttt{t_delete/1} e.g.,
  \[
  \texttt{t_insert\{?P[spouse->?S]\}}
  \]
- The T-logic hypothetical operators $\langle \rangle$ and $\sim\langle \rangle$ are supported in FLORA-2.
- This supports T-logic for non-tabled predicates, but what about tabled predicates? We can’t just reuse the answers in the table after an update.
- Update commands cannot occur in the scope of a tabled subgoal $S_T$ – if this were to happen, answers in $S_T$ might not all be satisfiable in the previous state or in the updated state.
Fortunately, XSB supports so-called incremental tables. Incremental tables ensure that tables that depend on an updated fact or facts are themselves updated.

The programmer can choose to update eagerly in a forward-chaining manner, or lazily when the potentially affected table is called.

In XSB, tables need to be explicitly declared as incremental, in $\text{FLORA-2}$ incrementality is declared at the module level.
An Approach to Defeasible Reasoning

As an illustration of what you can do with WFS, we consider an approach to defeasible reasoning [WGK+09]

- There are two user-defined predicates: opposes/2 and overrides/2.
- Two atoms $A_1$ and $A_2$ oppose each other if $A_1$ and $A_2$ cannot both be in the same model. opposes/2 is usually symmetric.
- An atom $A_1$ overrides an atom $A_2$ if the truth of $A_1$ in a model makes $A_2$ false in the model. overrides/2 is usually transitive.
Courteous Logic Programs

Assume each rule

\[ H :\neg \text{Body} \]

is rewritten as

\[ H :\neg \text{Body}, \neg \text{defeated}(H) \]

where defeated/1 is implemented as followed (this will look confusing at first :-)

\[ :- \text{table defeated/1.} \]
\[ \text{defeated}(A) :- \text{defeated_by}(A, _B). \]
\[ \text{defeated}(A) :- \text{defeats}(A, _B, A). \]
\[ \text{defeated_by}(A, B) :- \text{refutes}(B, A). \]
\[ \text{defeats}(A, B) :- \text{refutes}(A, B). \]
\[ \text{defeated_by}(A, B) :- \text{rebuts}(B, A). \]
\[ \text{rebuts}(A, B) :- \text{rebuts}(A, B). \]
\[ \text{refutes}(A, B) :- \text{conflicts}(A, B), \text{overrides}(A, B). \]
\[ \text{rebuts}(A, B) :- \text{conflicts}(A, B). \]
\[ \text{conflicts}(A, B) :- \text{opposes}(A, B), B. \]
Courteous Logic Programs

- Suppose you only specify \texttt{opposes(A,neg A)}, where \texttt{neg A} is the explicit negation of \texttt{A}, and \texttt{opposes/2} is symmetric.
- Since the predicate \texttt{overrides/2} is empty, the code of the last slide can be folded and reduced to:
  
  \begin{verbatim}
  :- table defeated/1.
defeated(A):- opposes(A,B),B. defeated(A):- opposes(B,A),B.
  \end{verbatim}

  which is the same as the semi-normal translation in [ADP95].
- As an example, using this defeasibility theory on the program \texttt{p neg p}.
  both \texttt{p} and \texttt{neg p} have the truth value undefined.
Courteous Logic Programs

- opposes/2 can of course be used for other purposes. For instance a rule such as
  
  ```prolog
  opposes(location(Obj,Loc1,Time),location(Obj,Loc2,Time)):-
  location(Loc1),location(Loc2),Loc1 \= Loc2.
  ```

  indicates that an object cannot be in two places at once.

- By specifying that one atom overrides another, you can have preference logic programming in the style of [CS02].

- Generally speaking, Subject Matter Experts (SMEs) and many KEs (Knowledge Engineers) will not want to devise their own argumentation theory, but will want to incorporate one. We’ll see later that this is the approach $\mathcal{FLORA-2}$ and Silk use.
Defeasibility of Rules

- Many defeasibility theories talk about opposition of rules as opposed to opposition of literals (e.g. [BE98]).
- If you want labelled rules, such as

  republican_rule: unconstitutional(Law):-
  imposes_mandate(Law).

  democratic_rule: constitutional(Law):-
  regulates_commerce(Law)

You can simply transform the labels into atoms in the body, so the atom-based approach is general.

- In this way, a rule may be *disqualified* if it rebuts itself, thus failing a derivation rather than making it undefined.
Defeasibility in Flora-2

- Flora-2 [YKWZ12] implements exactly this type of defeasibility.
- By “default” rules are strict – they do not have the defeated literal added to their body.
- Flora-2 supports 3 defeasibility theories
  - Original Generalized Courteous Logic: a rule $R$ is defeated if another rule refutes, cancels, or rebuts $R$.
  - New Generalized Courteous Logic: a rule $R$ is defeated if another rule $R'$ refutes, cancels, or rebuts $R$ and $R$ is itself not defeated.
  - Generalized Courteous Logic with Exclusion Constraints adds the ability to have more than two literals in opposition to each other: if $n$ literals oppose each other they may not all be true, but $n - 1$ of them could be true. For instance, a student might take 2 classes from a list, but not more than 2.
Defeasibility in Flora-2

Here is an example from the Flora-2 manual (using Prolog syntax). The underlying argumentation theory specifies that a rule is disqualified if it is cancelled.

\[
\begin{align*}
\text{device}(\text{printer}). & \quad \text{abused}(\text{bob},\text{printer}). \quad \text{pardoned}(\text{printer},\text{bob}). \\
\text{device}(\text{scanner}). & \quad \text{abused}(\text{bob},\text{scanner}). \quad \text{pardoned}(\text{scanner},\text{bill}) \\
\text{device}(\text{fax}). & \quad \text{abused}(\text{bill},\text{scanner}). \quad \text{abused}(\text{bill},\text{printer}). \quad \text{abused}(\text{mary},\text{fax}). \\
\text{person}(\text{bob}). & \quad \text{person}(\text{bill}). \quad \text{person}(\text{mary}).
\end{align*}
\]

@\{id1\}  \quad \text{authorized}(\text{Persn},\text{Device}) :- \\
\text{device}(\text{Device}), \quad \text{person}(\text{Persn}).

@\{id2(\text{Dev},\text{Persn})\} \quad \text{_cancel}(id1,\text{authorized}(\text{Persn},\text{Dev})) :- \\
\text{abused}(\text{Persn},\text{Dev}).

@\{id3\} \quad \text{_cancel}(id2(\text{Device},\text{Persn})) :- \\
\text{pardoned}(\text{Device},\text{Persn}).
**Benchmarks**

*Cyc*: A biological query to the Cyc ontology and reasoner, ported to Silk and compiled into XSB.

- CPU time 29.3
- There were 9,841,290 subgoals for tabled predicates, of which 1,781,628 created new tables, and 8,059,662 reused tables. The tables were sparsely answered, with 304,860 answers (12,447 of them redundant).
- So in this problem, XSB was creating about 60,000 new tables per second.
Benchmarks

**Haley**: Query to a temporal ontology constructed in Silk by Paul Haley.

- CPU time 0.21 seconds
- There were 126,917 subgoals for tabled predicates, of which 4947 created new tables, and 121,970 reused tables. Again, the tables were sparsely answered, with 1860 answers (948 of them redundant).
- So in this problem XSB was creating tables at a rate of about 30,000/sec.
Defeasible Reasoning

How to understand computations

- Prolog tracing is of limited help in computations like these.
  - Tabling’s suspend/resume mechanism breaks some of the conventions of Prolog’s 4-port debugger, but in general its at too low a level.
- In order to make defeasibility and other reasoning comprehensible, Silk has a sophisticated justification mechanism, extending [? , ?]. Justification is fully integrated with the Silk IDE.
- However justification is best for understanding inferences in a localized part of a program. How do you wrap your mind around a big tabled computation?
How to understand computations

- Like YAP, XSB has a `table_dump/[2,3,4]` predicate. Using this, the Silk user can at least view the tables at various levels of granularity.
- XSB also has a mechanism to write out its competition stack, either upon user request or before an exception is thrown. Using this mechanism with local evaluation users get a picture of the ancestors of the current goal and the various recursive components.
- Silk is also incorporating a new feature called *forest logging*. 
Recall that a tabled evaluation may be modeled as a forest of trees.

Our trace consists of logging (the important) tabling operations in a manner that can be queried or even visualized.
1. reach(1,Y) :- reach(1,Y)
2. reach(1,Y) :- edge(1,Z), reach(Z,Y)
3. reach(1,Y) :- reach(2,Y)
4. reach(2,Y) :- reach(2,Y)
5. reach(2,Y) :- edge(2,Z), reach(Z,Y)
6. reach(2,Y) :- reach(2,Y)
7. reach(2,Y) :-
8. reach(2,2) :-
9. reach(2,2) :-
10. reach(1,2) :-
11. reach(1,Y) :- reach(3,Y)
12. reach(3,Y) :- reach(3,Y)
13. reach(3,Y) :- edge(3,Z), reach(Z,Y)
14. reach(3,Y) :- reach(1,Y)
15. reach(3,2) :-
16. reach(3,Y) :- edge(3,Z), reach(Z,Y)
17. reach(3,1) :-
18. reach(1,Y) :- edge(1,Y)
19. reach(1,2) :-
20. reach(1,3) :-
21. reach(3,3) :-
22. reach(1,2) :-
23. reach(1,1) :-
24. reach(1,3) :-
25. reach(3,1) :-

:- table reach/2.
reach(X,Y) :- edge(X,Z), reach(Z,Y).
reach(X,Y) :- edge(X,Y).
edge(1,2) edge(1,3) edge(2,2) edge(3,1).
<table>
<thead>
<tr>
<th>Log File</th>
<th>Forest</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tc(reach(1,v0),null,new,0)</code></td>
<td>node 1</td>
<td></td>
</tr>
<tr>
<td><code>tc(reach(2,v0),reach(1,v0),new,1)</code></td>
<td>node 2</td>
<td>created by program clause resol.</td>
</tr>
<tr>
<td></td>
<td>node 3</td>
<td>created by program clause resol.</td>
</tr>
<tr>
<td><code>tc(reach(2,v0),reach(2,v0),incmp,2)</code></td>
<td>node 4</td>
<td>repeated subgoal registered</td>
</tr>
<tr>
<td></td>
<td>node 5</td>
<td>created by program clause resol.</td>
</tr>
<tr>
<td></td>
<td>node 6</td>
<td>created by program clause resol.</td>
</tr>
<tr>
<td><code>na([2],reach(2,v0),3)</code></td>
<td>node 7</td>
<td>registered as answer</td>
</tr>
<tr>
<td><code>ar([2],reach(2,v0),reach(2,v0),4)</code></td>
<td>node 8</td>
<td>created by answer resol.</td>
</tr>
<tr>
<td><code>cmp(reach(2,v0),2,5)</code></td>
<td>node 9</td>
<td><code>reach(2,v0)</code> completed</td>
</tr>
<tr>
<td><code>na([2],reach(1,v0),6)</code></td>
<td>node 10</td>
<td>created by return from completed table</td>
</tr>
<tr>
<td><code>tc(reach(3,v0),reach(1,v0),new,7)</code></td>
<td>node 11</td>
<td>created by program clause resol.</td>
</tr>
<tr>
<td><code>tc(reach(1,v0),reach(3,v0),incmp,8)</code></td>
<td>node 12</td>
<td>created by program clause resol.</td>
</tr>
<tr>
<td><code>ar([2],reach(1,v0),reach(3,v0),9)</code></td>
<td>node 13</td>
<td>repeated subgoal registered</td>
</tr>
<tr>
<td><code>na([2],reach(3,v0),10)</code></td>
<td>node 14</td>
<td>created by program clause resol.</td>
</tr>
<tr>
<td><code>na([1],reach(3,v0),11)</code></td>
<td>node 15</td>
<td>registered as an answer</td>
</tr>
<tr>
<td><code>na([3],reach(1,v0),12)</code></td>
<td>node 16</td>
<td>created by program clause resol.</td>
</tr>
<tr>
<td><code>ar([3],reach(1,v0),reach(3,v0),13)</code></td>
<td>node 17</td>
<td>created by program clause resol.</td>
</tr>
<tr>
<td><code>na([3],reach(3,v0),14)</code></td>
<td>node 18</td>
<td>(repeated answer)</td>
</tr>
<tr>
<td><code>ar([2],reach(3,v0),reach(1,v0),15)</code></td>
<td>node 19</td>
<td>created by program clause resol.</td>
</tr>
<tr>
<td><code>ar([1],reach(3,v0),reach(1,v0),16)</code></td>
<td>node 20</td>
<td>created by program clause resol.</td>
</tr>
<tr>
<td><code>na([1],reach(1,v0),17)</code></td>
<td>node 21</td>
<td>registered as an answer</td>
</tr>
<tr>
<td><code>ar([3],reach(1,v0),reach(3,v0),18)</code></td>
<td>node 22</td>
<td>created by answer return</td>
</tr>
<tr>
<td><code>ar([3],reach(3,v0),reach(1,v0),19)</code></td>
<td>node 23</td>
<td>created by answer resol.</td>
</tr>
<tr>
<td><code>na([2],reach(1,v0),18)</code></td>
<td>node 24</td>
<td>created by answer resol.</td>
</tr>
</tbody>
</table>
Forest Logging

The atoms on the previous slide have the following meaning:

- If a call to a tabled subgoal $S_1$ is made from a tree for $S_2$ the fact has the form $tc(S_1,S_2,\text{Stage},\text{Counter})$ is logged, where $\text{Counter}$ is the ordinal number of the fact, and $\text{Stage}$ is:
  - new if $S_1$ is a new subgoal
  - cmp if $S_1$ is not a new subgoal and has been completed
  - incmp if $S_1$ is not a new subgoal but has not been completed

- When a new answer $A$ is derived for subgoal $S$ and added to the table (i.e. $A$ is not already an answer for $S$) a fact of the form $na(A,S,\text{Counter})$ is logged.

- When an answer $A$ is returned to a consuming subgoal $S$ in a tree for $S_T$, a fact of the form $ar(A,S,S_T,\text{Counter})$ is logged. A log entry is made only if the table for $S$ is incomplete.

- **Subgoal completion**
  - When a set $S$ of subgoals is completed, a fact $cmp(S,\text{SCCNum},\text{Counter})$ is logged for each $S \in S$. $\text{SCCNum}$ is simply a number that can be used to group subgoals into mutually dependent sets of subgoal.
  - When a subgoal $S$ is *early completed*, a fact $cmp(S,\text{ec},\text{Counter})$ is logged.
Output is done wholly in C, so it adds about 30% overhead to a Flora execution.

The output is canonical, so loading the file can be very fast: on the order of 10,000,000 facts per minute.

Translating Prolog syntax back to Flora and Silk does slow things down somewhat. We’re still experimenting with how best to use this facility.
An aside on language extensions

- Some language extensions are simple syntactic sugar
  - Operators in Prolog
  - Basic User-Defined Functions in FLORA-2; Path expressions in FLORA-2

- Other extensions have a transformational semantics: the feature may be compiled into the underlying language, but the transformation is not be trivial or even straightforward.
  - FLORA-2 inheritance w.r.t. the well-founded semantics
  - Hilog in FLORA-2; Hilog in XSB
  - Cardinality constraints in ASP

- Finally, some extensions require changes in thinking about the semantics of the language (and fundamental changes in implementation)
  - Tabling in Prolog
  - Backtrackable updates in FLORA-2

- These distinctions are not hard and fast, and extensions may be useful at any level.
Other Features

- User-defined equality: $O_1 := O_2$
- Dynamic Subgoal Reordering
- Modules
## References I

<table>
<thead>
<tr>
<th>Reference</th>
<th>Authors</th>
<th>Title</th>
<th>Journal/Conference</th>
<th>Year</th>
</tr>
</thead>
</table>
References II

