Programming Knowledge with Frames and Logic

Part 2: Programming

3. Getting Around FLORA-2

Color Codes

- Black – what the user types
- Red – FLORA-2 prompt
- Green – FLORA-2 responses
- Blue – comments

Getting Started

- After installing:

  ./runflora *in Unix/Cygwin*

  .\runflora *in Windows*

  …*some chatter*…

  flora2 **?-**

- In Unix recommend putting this in .bashrc:

  alias flora='~/FLORA/flora2/runflora'

  assuming that FLORA-2 was installed in ~/FLORA

Compiling Programs

- Program files are expected to have the extension .flr
  - .flr doesn't need to be specified when compiling programs.
- The following will load and, *if necessary*, compile:
  - Load a file in the current directory

    flora2 **?-** [test].

    Or

    flora2 **?-** \load (test).
  - Load a file in /foo/bar/

    flora2 **?-** ['/foo/bar/test']. Windows: ['\\foo\\bar\\test']

    Or

    flora2 **?-** \load ('/foo/bar/test') .

    … *chatter* …

    flora2 **?-** *Now ready to accept commands and queries*

Temporary Programs

- Useful for quick tests
- Can write a program in-line and compile it

flora2 **?-** [ ]. // one underscore is *treated specially*

[FLORA: Type in FLORA program statements; Ctl-D when done]

a[b -> c].

…..

Ctl-D *in Unix*

Ctl-Z <*Return*> *in Windows/Cygwin*

… *chatter* …

flora2 **?-** *Now ready to accept commands and queries*

Asking Queries

- Once a program is loaded, you can start asking queries:

```
flora2 ?-    mary[works -> ?Where].

?Where = home

flora2 ?-
```

Important Commands at the FLORA-2 Shell

flora2 ?- \end. (or Ctl-D/Ctl-Z) *Drop into Prolog*

flora2 ?- \halt. *Quit* FLORA-2 **&** *Prolog*

- By default, FLORA-2 returns all solutions. Changing that:

  flora2 ?- \one**.**

  will start returning answers on-demand: typing “;” requests the next answer.

  flora2 ?- \all**.**

  revert back to the all-answers mode.
- \help – request help with the shell commands
- \demo(demoName). – compile and run a demo program

  (*Example*: flOneAll.flr)

Executing Queries at Startup

- At the Unix/Windows shell, one can request to evaluate an expression right after the FLORA-2 startup

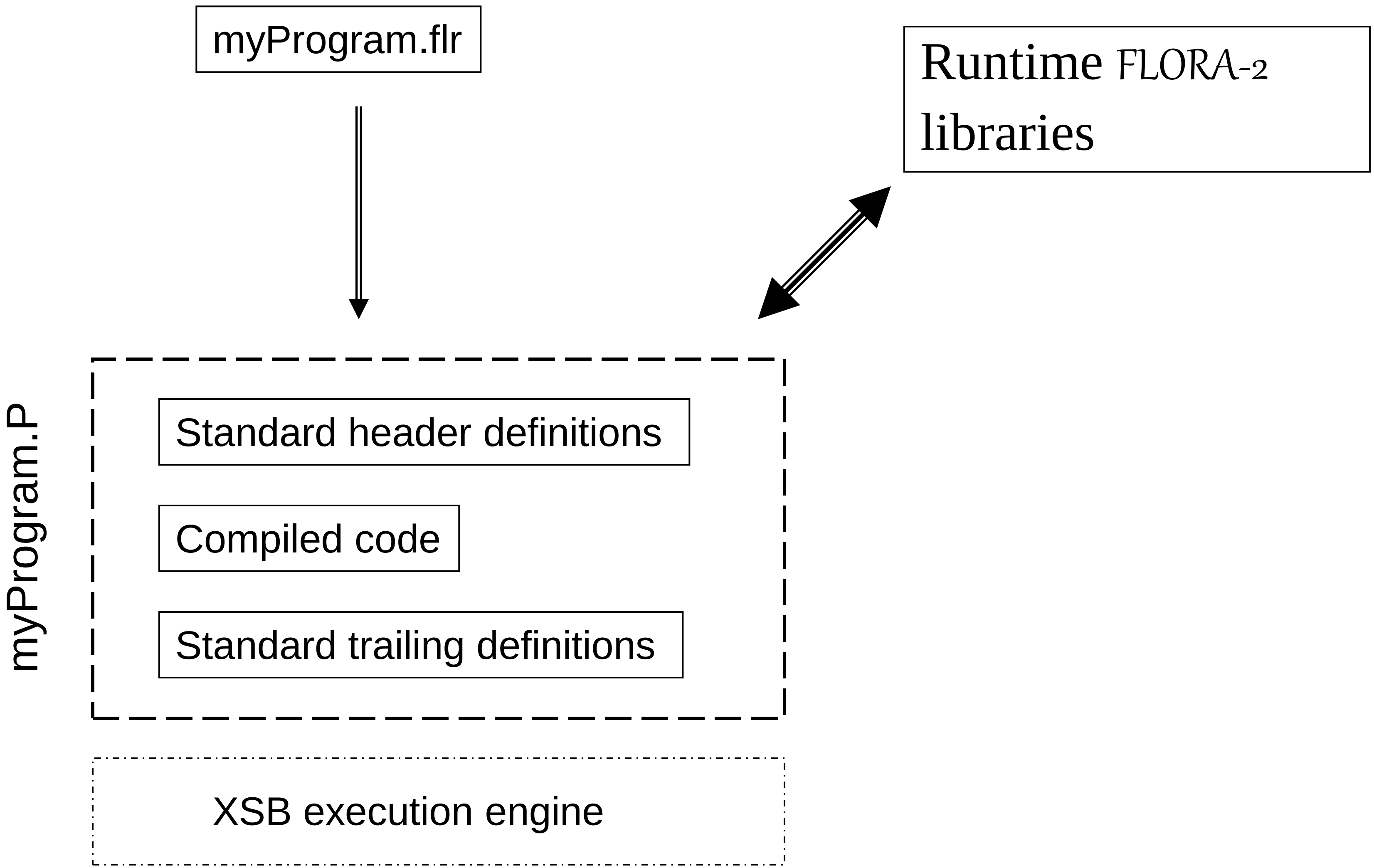
  ./runflora -e ″expression.″

  – Useful when need to repeat previous command repeatedly, especially for loading and compiling the same file over again:

  ./runflora -e ″\load(test).″

  (don't put spaces inside ″…″ (e.g., ″ \load (test).″ – some shell command interpreters have difficulty with them.)

How It Works


myProgram.flr
Runtime FLORA-2 libraries
myProgram.P
Standard header definitions
Compiled code
Standard trailing definitions
XSB execution engine

Variables

- Variables:
  - Symbols that begin with ?, followed by a letter, and then followed by zero or more letters and/or digits and/or underscores (e.g., ?X, ?name, ?v_5_)
  - ?_   or ? – *Anonymous* variable, a unique variable name is created.
    Different occurrences of ?_ and ? denote different variables
  - ?_*Alphanumeric* – *Silent* variable.
    Occurrences of the same variable within one rule denote the same variable.
    Bindings for silent variables are not returned as answers.
  - FLORA-2 does various checks and issues warnings for:
    - Singleton variables
    - Variables that appear in the rule head, but not in the rule body

  unless the variable is the ? or ?_ or a silent variable.

(*Example*: variableWarnings.flr)

Symbolic Constants and Strings

- Symbolic constants
  - If starts with a letter followed by zero or more letters and/or digits and/or underscores, then just write as is:

    a, *John*, *v_10)*
  - If has other characters then use single quotes: ′?AB #$ c′
- Strings
  - Lists of characters. Have special syntax:

    ″abc 12345 y″

    Same as [97,98,99,32,49,50,51,52,53,32,121]

Numbers, Comments

- Numbers
  – Integers: 123, 7895
  – Floats: 123.45, 56.567, 123E3, 345e-4

- Comments – like in Java/C++
  – // to the end of line
  – /* …milti-line comment… */

Methods and Cardinality Constraints

- FLORA-2 does not distinguish between functional and set-valued methods. All methods are set-valued by default.

  a[b1 -> c].

  a[b2 -> {c, d}].

- ***Cardinality constraints*** can be imposed on methods signatures to state how many values the method can have:

  A[M {2..4}=> D]. // *M can have 2 to 4 values of type D*

  – Functional (or scalar) method: cardinality constraint {0..1}

  C[m {0..1}=>b].

Logical Expressions

- Literals in rule bodies can be combined using , and ; (alternatively: *and* and *or*)

  head **:-** a, (b or c).

- Connectives , (and) and ; (or) can be used inside molecules:

  a[b -> c and d -> e ; f -> h].

  "," binds stronger than ";". The above is the same as

  a[b -> c, d -> e] ; a[f -> h].

- Negation is naf. Can be also used inside molecules:

  ?- a[not b -> c, d -> e ; f -> h].

Arithmetic Expressions

- FLORA-2 doesn't reorder goals. The following will cause a runtime error:

  ?- ?X > 1, ?X is 1 * (3+5).

  Make sure that variables are not used uninstantiated in expressions that don't allow this. Correct use:

  ?- ?X is 1 * (3+5), ?X > 1.

Modules

- Three types of modules:
  - FLORA-2 user modules (user programs)
    - Referred to with the @module idiom
  - FLORA-2 system modules (provided by the system)
    - Referred to with the @\module idiom (system module names start with a \)
  - Prolog (XSB) modules (Prolog programs: user-written or provided by XSB)
    - Referred to using the @\prolog or @\prolog(xsbmodule) idioms
      - @\prolog (abbr. @\plg) refers to the default XSB module or standard Prolog predicates
        - » E.g., …, writeln('Hello world')@\plg.
      - @\prolog(xsbmodule) (or @\plg(xsbmodule)) refers to XSB predicates defined in named XSB modules (hence need to know which XSB module each predicate belongs to)
        - » E.g., …, format('My name is ~w~n', [?Name])@\plg(format).

Modules: Dynamic Loading

- Program files are *not* associated with modules rigidly
  - Programs are ***loaded into modules*** at run time
  - Module is an abstraction for a piece of knowledge base
- **?-** [*myProgram* >> foobar].   *Or*

  ?- \load(*myProgram* >> foobar).

  myProgram.flr is loaded into module foobar.

  ?- [*anotherProgram* >> foobar].

  anotherProgram *replaces* *myProgram* in the module foobar.
  Can be done within the same session.

- [+*anotherProgram*>>foobar], \add *anotherProgram*>>foobar – *add* *anotherProgram* without erasing *myProgram*.

Default Module

- Default module is *main*:
  ?- [myProgram].

  Gets loaded into module *main*. Replaces whatever code or data was previously in that module.

Making Calls to Other Modules

- Suppose foobar is a module where a predicate p(?,?) and a method abc(?) -> … are defined.
- Calling these from within another module:

  head `:-` …, p(?X,f(a))@foobar, …, ?O[abc(123) -> ?Result]@foobar.
- Module can be decided at runtime:

  head `:-` …,?M=foobar, p(?X,f(a))@?M, …,?O[abc(123)->?Result]@?M.
- Modules can be queried: Which module has a definition for p(?,f(a))?

  `?-` p(?X,f(a))@?M.

Some Rules about Modules

- Module call cannot appear in a rule head. (Why?)
- Module references can be *grouped*:

  ?- (a(?X), ?O[b ->?W])@foo.
- Module references can be *nested*
  - *Inner overrides outer*:

    ?- (a(?X)@bar, ?O[b ->?W])@foo.
- \@ – special token that refers to the current module. If the following program is loaded into *foobar*, then

  a[b -> \@].

  ?- a[b -> ?X].

  binds ?X to *foobar*.

Useful Prolog Modules

- @\prolog(basics) – list manipulation, e.g., member/2, append/3, reverse/2, length/2, subset/2.
- @\prolog(format) – a (C-language) *printf* –like print statements.

FLORA-2 System Modules

- Provided by the system. Most useful are
  - @\sys – a bunch of system functions
    - abort(?Message)@\sys – abort execution (others later)
  - @\io – a bunch of I/O primitives
    - write(?Obj), writeln(?Obj), nl,
    - read(?Result)
    - see(?Filename), seen
    - tell(?Filename), told
    - File[exists(?F)]
    - File[remove{?F)]
    - Etc.
  - @\typecheck – defines constraints for type checking
    - ?- Cardinality[check(Mary[spouse=>?])]@\typecheck.
    - ?- Type[check(foo[?=>?], ?Violations)]@\typecheck.

Module Encapsulation

- Modules can be ***encapsulated*** to block unintended references
- By default, modules are *not* encapsulated
- If a module has an *export* directive then it becomes encapsulated
  - Only exported predicates or methods can be referenced by other modules
  - Predicates/methods can be exported to *specific modules* or to *all* modules
  - Predicates and methods can be exported as ***updatable***; default is non-updatable
  - Predicates/methods can be made encapsulated at run time (!) and additional items can be exported at run time

Export Statement

- Simple export:

  :- export{p(?,?), ?[foo -> ?]}.

  This exports to all modules.

  Note: use ?, not constants or other variables.
- Export to specific modules (*abc* and *cde*):

  :- export{(p(?,?) >> (abc, cde)), ?[foo -> ?]}.

  p/2 is exported only to abc and cde.

  foo -> is exported to all.
- Updatable export:

  :- export{p(?,?), updatable ?[foo -> ?]}.

  p/2 can be queried only; other modules can insert data for the method foo
- Exporting ISA/class membership:

  :- export {?:?, updatable ?::? >> abc}.

(*Example*: moduleExample.flr)

Dynamic Export

- All the previous statements can also be executed dynamically
  - If a module was not encapsulated it becomes encapsulated
  - Additional items can be exported at run time
- Examples of executable export statements:

  ?- export{p(?,?), ?[foo -> ?]}.

  ?- export{p(?,?), updatable ?[foo -> ?]}.

  ?- export{?:?, updatable ?::? >> abc}.

Multifile Modules

- Can split modules into multiple files and use the **#include** directive:

  #include ″foo.flr″ *relative path*

  #include ″/foo/bar/abc.flr″ *full path Unix*

  #include ″ \\foo\\bar\\abc.flr ″ *full path Windows*

- Note:
  - Must provide a complete relative or absolute name (with file extensions).
  - Must escape \ with another \ in Windows.
  - Can use Unix-style paths in Windows also.

Debugging

- Most common errors
  1. Mistyped variable
  2. Calling an undefined or unexported method/predicate (possibly due to mistyping)
  3. Suspicious program logic
  4. Wrong program logic

- 1–3 are handled by the compiler or the runtime environment
- 4 is handled by the trace debugger or other techniques (e.g., the venerable print statement)

Mistyped Variables

- Compiler warns about
  - Singleton variables
  - Variables in the rule head that don’t occur in rule body
- If such variables are intended, use anonymous or silent variables, e.g., ? or ?_abc. The compiler won’t flag those

(*Example*: variableWarnings.flr)

Mistyped or Undefined Methods/Predicates

- If a predicate/method was mistyped, it will likely be unique and thus undefined; the runtime catches those
- Undefinedness checks are turned off by default (for performance – about 50% slower)
- Enabling undefinedness checks:
  - Execute

    ?- Method[mustDefine(on)]@\sys.

    to turn on the checks in all modules.
  - Execute

    ?- Method[mustDefine(on,foobar)]@\sys.

    to turn on the checks in module foobar *only*
  - Can also turn off these checks wholesale or selectively

(*Example*: checkUndefined.flr)

Suspicious Program Logic

- A *tabled* predicate or method depends on a statement that produces a side effect:

  p(?X) **:-** …, write(?X)@\io, … .

- Possibly unintеded behavior:
  - 1st time:

    ?- p(hello).
    hello
    Yes
  - 2nd time:

    ?- p(hello).
    Yes
- Compiler will issue a warning. To block the warnings:

  :- ignore_depchk{%?@\io}. *Don't check dependencies on module flora(io)*

  Other forms:

  :- ignore_depchk{%foo(?)@?M}. *Don't check dependency on %foo(?) in any module*

  :- ignore_depchk{?[%abc(?,?) -> ?]}. *Don't check for %abc(?,?) -> in the current module*

(*Example*: tableVSnot.flr)

Debugger

- One can trace the execution of the program:

  ?- \trace. *Turn on interactive tracing*

  ?- \trace(file). *Noninteractive tracing. Put the trace into* file

  ?- \notrace. *Turn off tracing*

- How tracing works:
  - Shows which predicates are evaluated in which order
  - Which calls succeed and which fail
  - In interactive tracing:
    - <Return> – next step
    - S – trace non-interactively to the end; display everything
    - x – stop tracing the current call

Example of a Trace

```
?-    [].
a[b -> c].
aa[b -> f].
?X[m -> ?Y] :- ?Y[b -> ?X].
Ctl-D
?-    \trace.
?-    c[m -> ?Y].
  (2) Call:  c[m -> ?_h1281] ?  S
  (3) Call: (Checking against base facts) c[m -> ?_h1281]
  (3) Fail: (Checking against base facts) c[m -> ?_h1281]
  (4) Call: c[m -> ?_h1281]
  (4) Fail: c[m -> ?_h1281]
  (5) Call: ?_h1281[b -> c]
  (6) Call: (Checking against base facts) ?_h1281[b -> c]
  (6) Exit: (Checking against base facts) a[b -> c]
  (6) Redo: (Checking against base facts) a[b -> c]
  (6) Fail: (Checking against base facts) ?_h1281[b -> c]
  (7) Call: ?_h1281[b -> c]
  (7) Fail: ?_h1281[b -> c]
  (8) Call: ?_h1281[b -> c]
  (8) Fail: ?_h1281[b -> c]
  (5) Exit: a[b -> c]
  (5) Redo: a[b -> c]
  (5) Fail: ?_h1281[b -> c]
  (9) Call: c[m -> ?_h1281]
  (9) Fail: c[m -> ?_h1281]
  (2) Exit: c[m -> a]
  (2) Redo: c[m -> a] ? S
  (2) Fail: c[m -> ?_h1281]

?Y = a
```

(*Example*: trace.flr)

4. Low-level Details

HiLog vs. Prolog Representation

- Problem: FLORA-2's terms are HiLog; Prolog (XSB) uses Prolog terms – different internal representation
  - What if we want to talk to a Prolog program and pass arguments to it?

    Example: **?-** ?X=f(a), writeln(?X)@\prolog.

    flapply(f,a) <--- *not what we expected*

    ?X = f(a)
  - **Solution**: use a special primitive, p2h{?Prolog,?HiLog}

    Example: **?-** ?X=f(a), p2h{?P,?X}, writeln(?P)@\prolog.

    f(a) <--- *exactly what the doctor ordered*

    ?X = f(a)

  (*Example*: prologVShilog.flr)

  - **?-** ?X=f(a), writeln(?X)@\plgall(). <---- *also works*

To Table or Not to Table?

- Methods and predicates that start with a % are *assumed to produce side effects*
- Others are pure queries
  - *Pure queries*: p(?X,a), a[m -> ?X], ?X[p(a,b)]
  - *Side-effectful*: %p(?X,a), ?X[%p(a,b)]
- Only predicates and *Boolean* methods can have the % -prefix:
  - *Legal*: ?X[%p(a,b)]
  - *Not legal*: a[%m -> ?X]
- Pure queries are cached (implemented using XSB's tabled predicates); side-effectful predicates/methods are not cached.

(*Example*: tableVSnot.flr)

Why Table?

- *Queries* should use tabled methods/predicates
  – Recall that tabling implements the true logical semantics
  – Avoids infinite loops in query evaluation where possible
- When not to table:
  – ***Actions*** that have side effects (printing, changing the database state) should *not* be tabled.
    - This is a declarative way of thinking about the %-predicates and methods

5. Advanced Features

Type Checking

- Type correctness can be checked with an F-logic query:

```
type_error(?O,?M,?V) :-
              // value has wrong type
              (?O[?M ->?V],  ?O[?M =>?D])@?Mod,
             \naf ?V:?D@?Mod
             or
             // value exists, but type hasn't been specified
             (?O[?M  ->  ?V],  \naf ?O[?M  =>  ?D])@?Mod.

?-  type_error(?O,?M,?V).
```

Take out for semi-structured data

- If an answer exists then there is a type error. (Why?)
- There are also *standard methods to check types* (see manual: class Type in system module \typecheck)

Cardinality Checking

- The *type* system module defines constraints for checking cardinality
  - ?- Cardinality[check(?Obj[?Method=>?)]@\typecheck
    - If there are violations of cardinality constraints then ?Obj will get bound to the objects for which the violation was detected. For instance,

```
cl[foo {2..3}=> int].
c::cl.
o1:c.  o2:c.  o3:c.
o1[foo ->{1,2,3,4}].    c[foo->2].
o3[foo ->{3,4}].        cl[foo -> {3,4,5}].
```

Then the query

```
?- Cardinality[check(?O[foo=>?])]@\typecheck.
```

binds ?O to o1 and o2

- The system module *typecheck* has further elaborate methods for cardinality checking (see the manual)

Path Expressions

- A useful and natural shorthand
- ?X.?Y stands for the ?Z in ?X[?Y -> ?Z]

  For instance:

  a[b -> c].

  ?- a[b -> a.b].

  Yes

- Note: ?X.?Y denotes an object—it is not a formula

  But ?X.?Y[] is:

  ?X.?Y[] is true iff ?X[?Y -> ?] is true

Path Expressions (cont'd)

- ?X!?Y stands for a ?Z in ?X [|?Y -> ?Z|]

  ?X!?Y[] ≡ ?X [|?Y -> ?|]

- *What does* ?X.?Y!?Z *stand for*?

Path Expressions (cont'd)

- Path expressions can be combined with molecular syntax:

  ?X[m -> ?Z].?Y.?Z [abc -> ?Q]

 is:

  ?X[m -> ?Z], ?X[?Y -> ?V], ?V[?Z -> ?W], ?W[abc -> ?Q]

 Or, in one molecule:

  ?X[m -> ?Z, ?Y -> ?V[?Z -> ?W[abc -> ?Q]]]

Nested Molecules

- Nested molecules are broken apart (as we have seen)
- But what is the ordering? – important since evaluation is left-to-right

- Molecules nested inside molecules:

  a[b -> c[d -> e]]

  breaks down as a[b -> c], c[d ->e].

  But a[b[c -> d] -> e]

  as b[c -> d], a[b ->e]

- Molecules nested inside predicates:

  p(a[b -> c]) *breaks down as* p(a), a[b ->c]

  p(a.b) *breaks down as* a.b=?X, p(?X) (Why?)

  p(a.b[]) *breaks down as* p(?X), a[b ->?X]

  (*Example*: molBreak.flr)

- *What does the following mean*?

  a[b -> c][d -> e]

Nested *Reified* Molecules

- Don't confuse

  p(a[b -> c]) and a[b -> c[d -> e]]

  with ***reified*** nested molecules:

  p(${a[b -> c]}) and a[b -> ${c[d -> e]}]

- What are the latter broken down to?

Aggregate Expressions

- Like in SQL, but better:
  - Can evaluate subquery and apply sum/count/avg/… to the result
  - Can group by certain variables and then apply sum/count/… to each group
  - Can create sets or bags, not just sums, counts, etc.

Aggregate Expressions: Syntax & Semantics

- General syntax:

  ?Result = *aggFunction*{AggVar[GroupingVars] | *Query*}

- *aggFunction*:
  - *min, max, count, sum, avg* – the usual stuff
  - *setof* – collects list of values, duplicates removed
  - *bagof* – same but duplicates remain
- aggVar – <u>single</u> variable, but not a limitation
  - Can do something like avg{?X | *query(?Y)*, ?X is exp(?Y+1,2)} or setof{?X | …, ?X = f(?Y,?Z)}
- GroupingVars – comma-separated list of vars on which to group (like SQL's GROUP BY)
- Returns *aggFunction* applied to the list(s) of AggVar (grouped by GroupingVars) such that *Query* is satisfied

Aggregate Syntax & Semantics (cont'd)

- Aggregates can occur where a number or a list can – hence can occur in expressions

```
?- ?Z=count{?Year| john.salary(?Year) < max{?S| john[salary(?Y2) ->?S], ?Y2< ?Year} }.
```

- What if *Query* in the aggregate returns nothing?
  - *sum*, *avg*, *min*, *max*, *count*: will fail (are false)
  - *setof*, *bagof*: return empty list

(*Example*: aggregate.flr)

Aggregates and Set-valued Methods

- Convenient shortcuts for collecting results of a method into a list

  ?O[?M ->-> ?L] – ?L is the list of elt's such that ?O[?M -> elt] is true

  Same as ?setof{?X| ?O[?M -> ?X]}

  ?O[|?M ->-> ?L|] – ?L is the list of elt's such that ?O[|?M -> elt|] is true

  Same as ?L=setof{?X| ?O[|?M -> ?X|]}

- Set containment

  ?O[?M +>> ?S] – true if ?S is a list & $\forall$ s $\in$?S, ?O[?M ->s] is true

  ?O[|?M +>> ?S|] – true if ?S is a list & $\forall$ s $\in$?S, ?O[?M ->s] is true

Anonymous OIDs (Skolem Constants)

- Like blank nodes in RDF (but with sane semantics)
- Useful when one doesn’t want to invent object IDs and relies on the system (e.g., individual parts in a warehouse database could use this feature)
- Can be numbered or unnumbered
  - Unnumbered: \# – *different* occurrences mean *different* IDs:

    \#[name ->'John', spouse -> \#[name ->'Mary']]

  - Numbered: \#1, \#2, \#3, … – different occurrences of, e.g., \#2 *in the same clause* means the same ID:

    \#1[name ->'Jay', spouse -> \#[name ->'Ann', spouse -> \#1]].

    Same ID

    \#1 [name -> 'Jay'].

    \#1 [name -> 'Jay'].

    Different IDs

Anonymous OIDs (cont'd)

- \#, \#1, \#2, etc., are plain symbols. Can use them to construct terms. For instance: \#(\#1,\#,\#2,\#1)

  \#1:student[name -> 'Joe',
  advisor -> {\#(\#1)[name ->'Phil'],
  \#(\#1)[name -> 'Bob'] }].

  – *Why is this useful*?

- \#, \#1, … can appear *only* in the facts and rule heads.

  ?- a[m -> \#].

  – *Why does such a query make no sense*?

Equality

- Sometimes need to be able to say that two things are the same (e.g., same Web resource with 2 URIs)
- FLORA-2 has the **:=:** predicate for this. For instance:

  a **:=:** b.

  p(a).

  ?- p(b).

  Yes

- Well, not so fast…
  - Equality maintenance is computationally expensive, so it is off by default
  - Can be turned on/off on a per module basis
  - Different types of equality: *none*, *basic*
  - Has some limitations

Types of Equality

- *none* – no equality maintenance

  `:=:` is like =.

- *basic* – the usual kind of equality

Enabling Equality

- At compile time:

  `:-` setsemantics{equality(basic)}.

- At run time:

  `?-` setsemantics{equality(none)}

  – Can be set and reset at run time

- Can find out at run time what kind of equality is in use:

  `?-` semantics{equality(?Type)}.

  ?Type=none

(*Example*: equality.flr)

Limitations of Equality Maintenance in FLORA-2

- Congruence axiom for equality:
  - a=b ∧ φ[a] implies φ[b]
  - This is very expensive
- FLORA-2 uses ***shallow*** congruence:
  - Does substitution only at levels 0 and 1:
    - p :=: q, p(a) implies q(a) *level 0*
    - a :=: b, p(a) implies p(b) *level 1*
    - a :=: b, a[m -> v] implies b[m -> v].
    - v :=: w, a[m -> v] implies a[m -> w].
    - But: a :=: b, p(f(a)) does *not* imply p(f(b)) *level 2*

Avoiding Equality

- In many cases, equality is too heavy for what the user might actually need.
- Try to use the preprocessor instead:

```
#define  w3   "http://www.w3.org/"
 ?-  w3[fetch -> ?Page].
```

Data Types

- **URI data type:** "…"^^\iri (IRI stands for International Resource Identifier, a W3C standard)

  e.g., "http://www.w3.org"^^\iri

  – Compact IRIs
    - Can define ***prefixes*** and then use them to abbreviate long URIs

```
:- iriprefix{W3 = 'http://w3.org/'}.

s(?X) :- ?X[a -> W3#abc].  // W2#abc expands to "http://w3.org/abc"^^\iri
```

- Standard methods exist to extract the *scheme*, *user*, *host*, *port*, *path*, *query*, and *fragment* parts of IRIs

Data Types (contd.)

- Date and Time type
  - ”2007-01-21T11:22:44+05:44”^^\dateTime (or ^^\dt)

    +05:44 is time zone

    ”2007-02-11T09:55:33”^^\dateTime or

    ”2007-03-12”^^\dateTime
  - Methods for extracting parts:
    - *\year*, *\month*, *\day*, *\hour*, *\minute*, *\second*, *\zoneSign*, *\zoneHour*, *\zoneMinute*
- Time type
  - ”11:29:55”^^\time (or ^^\t)
  - Methods: *\hour*, *\minute*, *\second*
- Comparison and arithmetic operations for date and time are supported (can add/subtract duration types)
- Other data types also exist

Control Constructs

- **\if** (*cond*) **\then** (*then-part*) **\else** (*else-part*)
- **\if** (*cond*) **\then** (*then-part*)
  – Important difference with Prolog: if *cond* is false, **if-then** is still true, but the *then-part* is not executed
- **\unless** (*cond*) **\do** (*unless-part*)
  – Execute the *unless-part* if *cond* is false
  – If *cond* is true, do nothing (but the whole **unless-do** statement is true)
- Has also while/until loops

Metaprogramming

- FLORA-2 allows variables everywhere, so much of the meta-information can be queried
- The reification operator allows one to construct arbitrary facts/queries, *even rules*:

  ?- p(?X), q(?Y), ?Z= ${?X[abc **->** ?Y]}.

  ?- ?X[abc −> ?Y].

  ?- ?X = ${a **:-** b}, ….
- What is missing?
  - The ability to retrieve an arbitrary term and find out what kind of thing it is
  - Whether it is a term or a formula
  - What module it belongs to?

Meta-unification

- This capability is provided by the ***meta-unification*** operator, ~
- Not to be confused with the regular unification operator, =
- Examples:

```
?- a[b -> ?Y]@foo ~ ?X@?M.
?X = ${a[b -> ?Y]@foo}
?M = foo

?- a[b -> ?Y] ~ ?X[?B -> c]@?M.
?B = b
?M = main
?X = a
?Y = c
```

Meta-unification (cont'd)

- When both the module and the type of formula is known, then "=" will do:

```
?- ${?X[a -> b]@foo} = ${o[?A -> ?B]@foo}.
```

But this will fail:

```
?- ${?X[a -> b]@?M} = ${o[?A -> ?B]@foo}.
No
```

"=" will work in many cases, but use ~ when in doubt:

```
?- ${?X[a -> b]@?M} ~ ${o[?A -> ?B]@foo}.
?X = o
?M = foo
?A = a
?B = b
```

Recognizing Unknown Meta-terms

- ?X ~ (?A, ?B) A conjunction (= also ok)
- ?X ~ (?A; ?B) A disjunction (= ok)
- ?X ~ ?Y@?M A molecule or a HiLog formula
- ?X ~ ? [? –> ?] A functional molecule
- … … …

6. Updating the Knowledge Base

What Kinds of Updates?

- In FLORA-2, the knowledge base can be changed in the following ways:
  - Insert/delete facts in a module
  - Insert/delete rules in a module
  - Create a completely new module on-the-fly (at run time) and put data and rules into it
    - E.g., create a new agent dynamically

Adding and Deleting Facts

- Support provided for
  - Non-logical updates, which only have operational semantics (like in Prolog, but more powerful) – ***non-backtrackable*** and thus ***non-transactional*** updates
  - Logical updates as in Transaction Logic – ***transactional*** updates
- Non-transactional: *insert*, *delete*, *insertall*, *deleteall*, *erase*, *eraseall*
- Transactional: *t_insert*, *t_delete*, *t_insertall*, *t_deleteall*, *t_erase*, *t_eraseall* (synonyms: *tinsert*, *tdelete*, etc.)

Syntax of Update Operators

- *updateOp*{ *Literals* }
- *updateOp*{ *Literals* | *Query* }
- *Literals*: stuff to delete
- *Query*: condition on *Literals*
- The exact meaning of *Literals* and *Query* depends on the particular *updateOp*

Insert Operators (non-logical)

- Unconditional:

  ?- p(?X), q(?Y), insert{ ?X[has -> ?Y] }.

  inserts ?X[has -> ?Y] *for the binding of* ?X *and* ?Y

  ?- p(?X), q(?Y), insert**all**{ ?X[has -> ?Y] }.

  no difference in this context

- Conditional:

  ?- \one. *To prevent backtracking*

  ?- p(?X), insert{ ?X[has -> ?Y] | q(?Y) }.

  insert for some ?Y *such that* q(?Y) *is true*

  ?- p(?X), insertall{ ?X[has -> ?Y] | q(?Y) }.

  insert for all ?Y *such that* q(?Y) *is true*

Delete Operators (non-logical)

- Unconditional

  ?- \one.

  ?- q(?X), delete{p(?X,?Y), a[b ->?Y]}. *Delete for some ?Y*

  ?- q(?X), deleteall{p(?X,?Y), a[b ->?Y]}. *Delete for all ?Y*

- Conditional

  ?- \one.

  ?- q(?X), delete{p(?X,?Y) | a[b ->?Y]}. *Delete for some ?Y*

  ?- q(?X), deleteall{p(?X,?Y) | a[b ->?Y]}. *Delete for all ?Y*

(*Example*: delete.flr)

Delete Operators (cont'd)

- erase{fact1, fact2,…}
  – Works like delete, but also deletes all objects reachable from the specified object
- eraseall{facts|query}
  – Works like deleteall, but for each deleted object also deletes the objects that are reachable from it

(*Example*: erase.flr)

Transactional (Logical) Updates

- The basic difference is that a postcondition can affect what was inserted or deleted
  - Non-logical:

    ?- insert{p(a)}, deleteall{q(?X)}, a[b ->c].

    p(a) will be inserted / q(?X) deleted *regardless* of whether a[b ->c] was true or false
  - Logical:

    ?- t_insert{pp(a)}, t_deleteall{qq(?X)}, a[b ->c].

    updates will be done only if a[b ->c] remains true after

KB Updates and Tabling

- Program:

  q(a).

  p(?X) :- q(?X).

  ?- p(a).

  Yes

  ?- p(b).

  No

  ?- delete{q(a)}, insert{q(b)}.

  Yes

- Problem:

  ?- p(a). Yes/No?

  Yes

  ?- p(b). Yes/No?

  No.

  Why the wrong answers?

Because p(?) *is tabled and the old answers have been tabled!*

Updates and Tabling (cont'd)

- The problem of updating tables when the underlying data changes is similar to (but harder than) the problem of updating materialized views in databases
  - *Should be handled in the XSB engine and is hard*
- FLORA-2 helps the user get a handle on this problem by:
  - Automatically taking care of updating tables for the facts
  - Providing the refresh{…} primitive to let the user selectively clean up tabled data
  - Providing a sledge hummer: the Tables[abolish]@\sys method

Abolishing Tables

- Tables[abolish]@\system: clears out all tables
  - Previous queries would have to be recomputed – performance penalty
  - ***Cannot*** be called during a computation of a tabled predicate or molecule – XSB will coredump!
- refresh{fact1, fact2,…}: selectively removes the tabled data that *unifies* with the specified facts (facts can have variables in them
  - Lesser performance penalty
  - Can be used in more cases: refresh{…} will crash XSB only if you call it while computing the facts being refreshed

(*Example*: refresh.flr)

Tabled Literals that Depend on Updates

- If a tabled literal depends on an update, then executing it twice will execute the update only once – *probably an error in the program logic*
- FLORA-2 will issue a warning
- To block the warning (if the logic is correct), use

  :- ignore_depchk{*skeleton*, *skeleton*, …}.

  The skeletons specify the predicates that tabled predicates can depend on without triggering the warning

  - Warnings are triggered for insert/delete ops, any predicate or method that starts with a %.

(*Example*: depchk.flr)

Updates and Meta-Programming

- Update operators can take variables that range over formulas – ***metaupdates***
- Module foo:

```
%update(?X,?Y) :-    // check that args have the right form
                   ?X ~ ?O[?M -> ?], ?Y ~ ?O[?M -> ?],
                   !,
                   delete{?X},  insert{?Y}.
%update(?_X,?_Y) :-  abort([?Y, ' not updating ', ?X])@\sys.
```

- Module main:

```
?- %update(${a[b -> ?]}, ${a[b -> d]})@foo.
```

Inserting/Deleting Rules

- Useful when knowledge changes dynamically
- Especially for creation of new agents and stuffing them with rules
- FLORA-2 rules can be ***static*** or ***dynamic***
- *Static rules*:
  - Those that you put in your program; they can't be deleted or changed
- *Dynamic rules*:
  - Those that were inserted using insertrule_a{…} or insertrule_z{…} primitives; they can be deleted using deleterule{…}

Rule Insertion Operators

- insertrule_a{ (rule1), (rule2),…}
  - Inserts the rule(s) *before* all other rules (static or dynamic)

    ?- insertrule_a{p(?X) **:-** ?X[a -> b]}.

    ?- insertrule_a{(a **:-** b), (c **:-** d)}
- insertrule_z { (rule1), (rule2),…}
  - Inserts the rules *after* all other rules (static or dynamic)

    ?- insertrule_z{p(?X) **:-** ?X[a -> b]}.
- Note: static rules are always stuck in the middle of the program

Insertion of Rules into Another Module

- If *foobar* is another module:

  ?- insertrule_z{(a **:-** b)@foobar, (c **:-** d)@foobar}

- Module may already exist or be created on-the-fly:

  ?- newmodule{foobar}.

  If foobar does not exist, it will be created "empty"

Rule Deletion Operator

- Only previously inserted rules (i.e., dynamic rules) can be deleted
- Operator: deleterule{ (rule1), (rule2), …}
  - Delete every dynamic rule that *matches* rule1, rule2, …
- insertrule_a, insertrule_z, deleterule are *non*-transactional (so not backtrackable).
  - But this is unlikely to matter: Who would stick a post-condition after insertrule/deleterule? (Someone *too* sophisticated.)

Flexible Deletion

- The rules in deleterule can be more flexible than what is allowed in insertrule and in static rules:
  - Can have variables in the rule head & body
- Examples:

  ?- deleterule{?H :- ?X[abc -> ?Y]}.
  - Delete every dynamic rule with the body that looks like ?X[abc -> ?Y]

  ?- deleterule{?X[abc -> ?Y] :- ?B}.
  - Delete every dynamic rule with the head that looks like ?X[abc ->?Y]

  ?- deleterule{(?H :- ?B@?M)@?N}.
  - Delete *every* dynamic rule in every module!

  (*Example*: dynrules.flr)

7. Future Plans

Research Issues

- Speed up query evaluation
- Approximate reasoning
- Better implementation of transactional updates
- Implementation of Concurrent Transaction Logic

Problems that Need XSB Work

- Cuts over tabled predicates

Questions?