Programming Knowledge with Frames and Logic

Michael Kifer

Stony Brook University

Part1: Foundations

What’s in This Tutorial?

Part 1: Foundations

1. Introduction
2. Background
   - F-logic (Frame Logic)
   - HiLog
   - Transaction Logic
   - Top-down execution and tabling

What’s in This Tutorial?

Part 2: Programming

3. Getting Around FLORA-2
   - Getting started
   - Modules
   - Multifile modules
   - Debugging
4. Some Low-level Details
   - HiLog vs. Prolog representation of terms
   - To table or not to table?

What’s in This Tutorial?

5. Advanced Features
   - Path expressions
   - Aggregates
   - Anonymous OIDs
   - Equality
   - Control constructs
   - Metaprogramming
6. Updating the Knowledge Base
   - Non-logical updates
   - Logical updates
   - Limitations
   - Inserting and deleting rules
7. Future plans

1. Introduction

What’s Wrong with Knowledge Representation Based on Classical Logic?

- Essentially flat data structures:

  person(John, ‘123 Main St.’, 34)

- Awkward meta-programming:

  Which predicates mention John?

- Ill-suited for modeling side effects:

  State changes, I/O

A Solution

- Flat data structures:

  Frames (F-logic)

- Awkward meta-programming:

  Higher-order syntax (HiLog + F-logic)

- Modeling side effects:

  Logic of updates (Transaction Logic)

What is FLORA-2 ?

- **F**-**L**ogic t**RA**nslator
- Realizes the vision of logic-based KR with frames, meta, and side-efects. Founded on
  - F-logic
  - HiLog
  - Transaction Logic
- Practical & usable KR and programming environment
  - Declarative
  - Object-oriented
  - Logic-programming style
  - Overcomes most of the usability problems with Prolog

What is FLORA-2 ?

- Builds on earlier experience with implementations of F-logic:
  - FLORID, FLIP, FLORA-1 (which don't support HiLog & Transaction Logic)
- Differs in spirit from other F-logic based systems
  - FLORID, Ontobroker are *query languages;* cannot live without a procedural language (C++, Java)
  - FLORA-2 is a complete *programming language*; can be used in the query language capacity as well.
- http://flora.sourceforge.net
- A recent overview: [Yang, Kifer, Zhao, ODBASE-2003]

Applications of FLORA-2

- Ontology management
- Knowledge-based networking
- Information integration
- Software engineering
- Agents
- Anything that requires manipulation of complex structured (especially semi-structured) data

Other F-logic Based Systems

- *?????* (U. Melbourne – M. Lawley) – early 90’s; first Prolog-based implementation
- *FLORID* (U. Freiburg – Lausen et al.) – late 90’s; the only C++ based implementation
- *FLIP* (U. Freiburg – Ludaescher) – late 90’s; first XSB based implementation. Inspired the FLORA effort
- *TFL* (Tech. U. Valencia – Carsi) – late 90’s; first attempt at F-logic + Transaction Logic
- *SILRI* (Karlsruhe – Decker et al.) – late 90’s; Java based
- *TRIPLE* (Stanford – Decker et al.) – early 2000’s; Java
- *OntoBroker* (Ontoprise.de, now Semafora) – 2000; commercial

2. Background

Desirable Background Knowledge

- Predicate calculus
  - Good understanding of its model theory
- Logic programming/Deductive databases
  - Bottom-up execution (T_P operator)
  - Top-down execution (SLD resolution)
  - Negation as failure / Well-founded negation
- Prolog language

2.1. Background: F-Logic

Basic Ideas Behind F-logic

- Take complex data types as in object-oriented databases
- Combine them with logic
- Use the result as a programming language

What F-Logic Provides

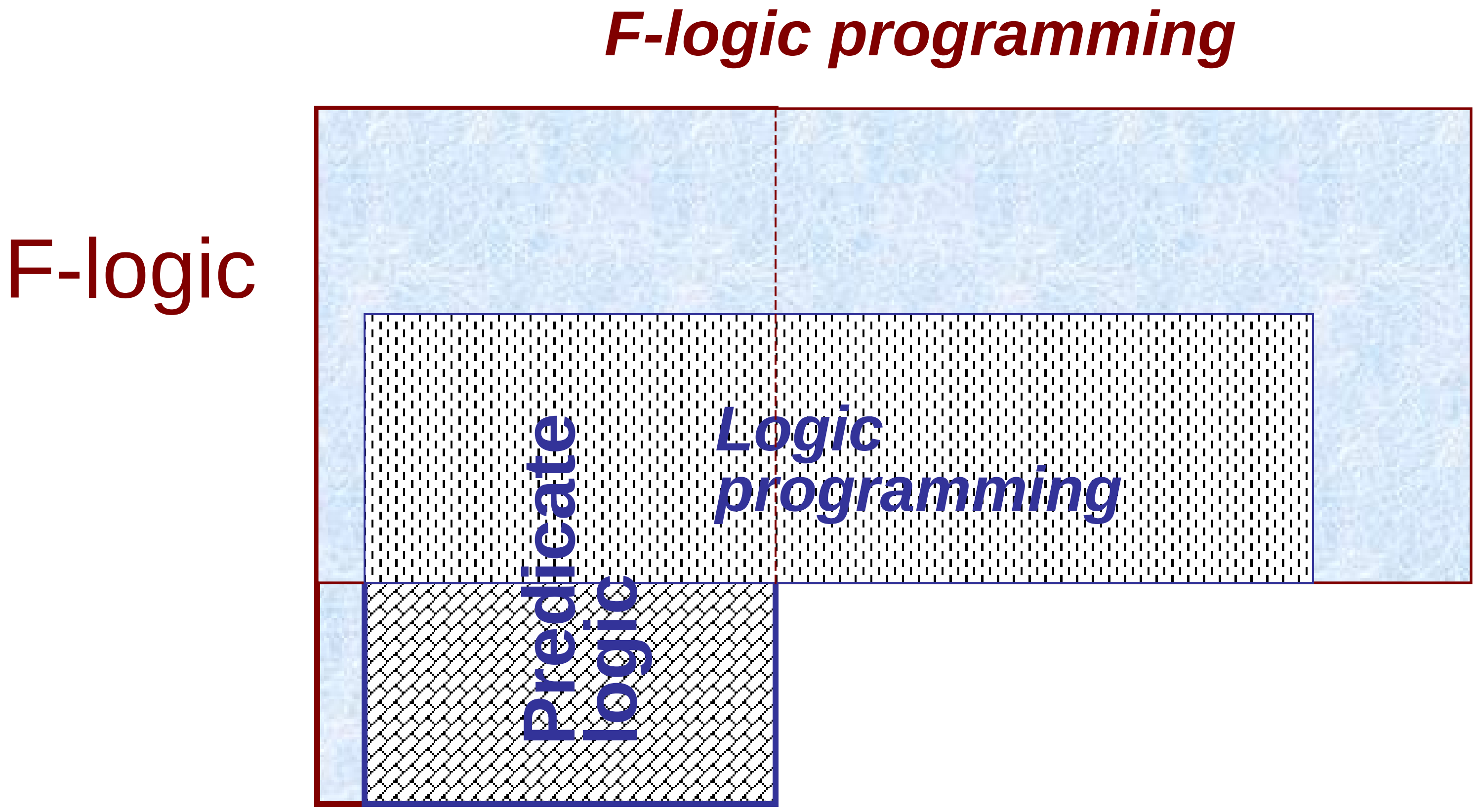
- Objects with complex internal structure
- Class hierarchies and inheritance
- Typing
- Encapsulation
- Background:
  - Basic theory: [Kifer & Lausen SIGMOD-89], [Kifer,Lausen,Wu JACM-95]
  - Path expression syntax: [Frohn, Lausen, Uphoff VLDB-84]
  - Semantics for non-monotonic inheritance: [Yang & Kifer, ODBASE 2002]
  - Meta-programming + other extensions: [Yang & Kifer, ODBASE 2002]

Relationship to Standard Logic

$$\frac{\text{O-O programming}}{\text{F-logic}} = \frac{\text{Relational programming}}{\text{Predicate calculus}}$$

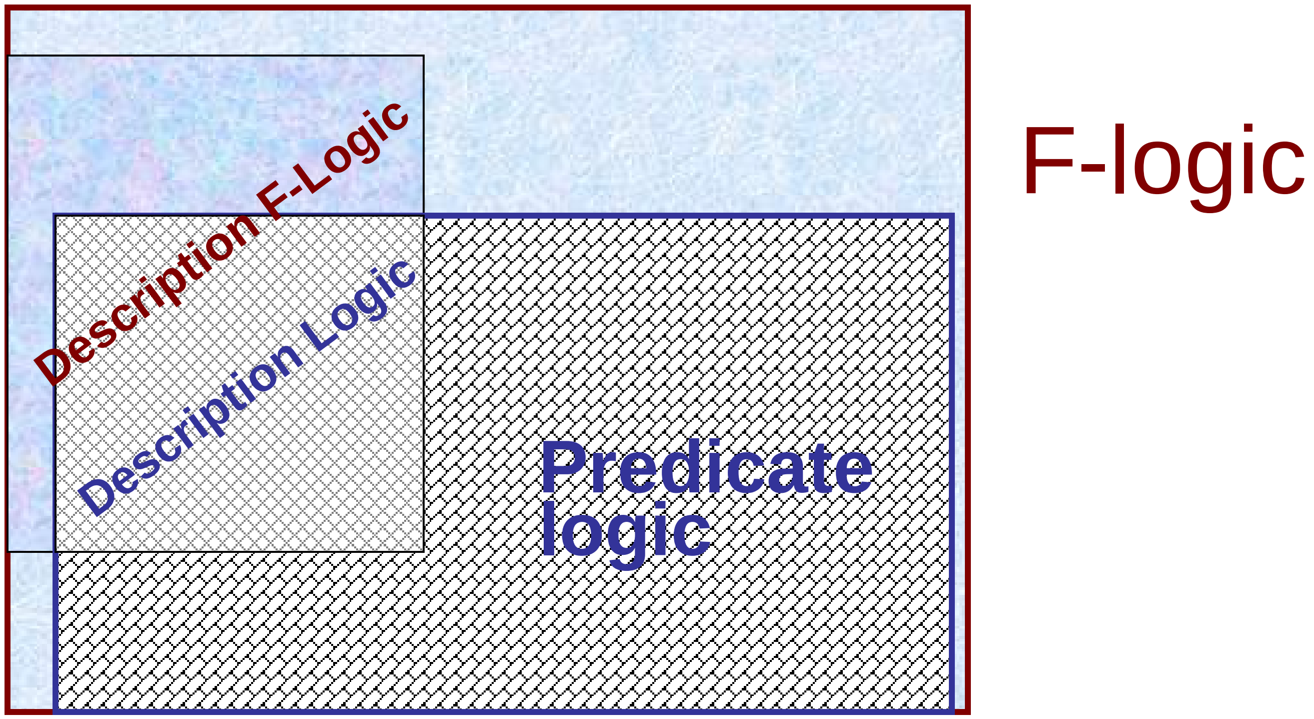
Relationship to Standard Logic (cont'd)

First-order flavor vs. logic programming flavor.



Relationship to Description Logic

A description logic subset can be developed in F-logic
[Balaban 1995, The F-logic Approach for Description Languages]



F-logic: Simple Examples

Object Id

Attribute

Object description:

John[*name* -> 'John Doe', *phones* -> {6313214567, 6313214566}, *children* -> {Bob, Mary}]

Mary[*name* -> 'Mary Doe', *phones* -> {2121234567, 2121237645}, *children* -> {Anne, Alice}]

Attribute

Structure can be nested:

Sally[*spouse* -> John[*address* -> '123 Main St.']]

Examples (cont’d)

- Historic notes:
  - The original F-logic distinguished between functional (`->`) and set-valued (`->>`) attributes
    - In FLORA-2 this has been simplified and generalized:
      - Only set-valued methods and only `->` are used
      - Can specify cardinality constraints. The constraint {0:1} corresponds to functional attributes
  - In F-logic, variables were denoted by capitalized symbols
    - In FLORA-2 variables are preceded with a ?.
    - Constants can start with lowercase or uppercase – does not matter:
      - John, betty.

Examples (contd.)

ISA hierarchy:

John : Person - *class membership*
Mary : Person
alice : Student

Student :: Person - *subclass relationship*

Class & instance at the same time

Student : EntityType
Person : EntityType

Examples (Contd.)

Methods: like attributes, but take arguments

?P[*ageAsOf*(?Year) -> ?Age] :-
    ?P:Person, ?P[*born* -> ?B], ?Age \is ?Year–?B.

- Attributes can be viewed as methods with no arguments

Query:

John's children who were born when he was 30+ years old:

?- John[*born* -> ?Y, *children* -> ?C],
    ?C[*born* -> ?B], ?B > ?Y+30.

or

?- John[*ageAsOf*(?Y) -> 30, *children* -> ?C],
    ?C[*born* -> ?B], ?B>?Y.

Examples (Contd.)

- **Type signatures**: Define the types for method arguments and for their results

```
Person[born => \integer,
       ageAsOf(integer) => \integer,
       name => \string,
         address => \string,
       children =>  person].
```

- Signatures can be queried:

```
        ?-  Person[name => ?Type].
Answer:    ?Type = \string
       ?-  Person[?Attr => \string].
Answer:     ?Attr = name
            ?Attr = address
```

Note: builtin types, like \integer, start with a backslash.

Syntax

- Object ids:
  - Terms like in Prolog, but constants, functions can be capitalized – John, abc, f(john,34), Car(red,20000)
  - Below, O, C, M, T, ... denote usual first order terms
- IsA hierarchy (*isa-atoms*):
  - O:C -- object O is a ***member*** of class C
  - C::S -- C is a ***subclass*** of S
- Structure (*object-atoms*):
  - O [*Method* -> Value] -- invocation of method
- Type (*signature-atoms*):
  - Class [*Method* => Class] – a method signature
- Combinations of the above:
  - and, or, negation, quantifiers

More Examples

Browsing IsA hierarchy:

```
?-  John : ?X.
?-  Student ::?Y
```

Virtual (view) class:

```
?X : Redcar  :-  ?X:Car , ?X[color  ->  red].
```

and

Meta-query about schema:

```
?O[attributesOf(?Class) -> ?A]  :-
        ?O[?A ->?V],  ?V:?Class.
```

Rule defines method, which returns attributes whose range is class Class
α :- β is implication, α or $\neg\beta$

Parameterized family of classes:

```
[]:list(?T).
[?X|?L]:list(?T) :-  ?X:?T,  ?L:list(?T).
```

E.g., list(integer), list(student)

Model Theory for Object Definitions

Simplified (so-called *Herbrand*) semantics:

Universe: HB – set of all variable-free terms (“ground” terms)

Interpretation: **I** = (HB,$I_{->}$,∈,<)

where < : partial order on HB

∈ : binary relationship on HB

$I_{->}$: $\underbrace{HB}_{\text{methods}} \rightarrow (\underbrace{HB}_{\text{objects}} \xrightarrow{partial} \underbrace{powerset(HB)}_{\text{values}})$

Satisfaction of formulas in I:

I |= o[*m*->v] if v ∈ $I_{->}$(*m*)(o)

I |= o:c if o ∈ c

I |= c::s if c < s

Model Theory for Types

Added

Interpretation: $\mathbf{I} = (HB, I_{\text{->}}, \in, <, I_{\text{=>}})$

where $I_{\text{=>}}$: $\boxed{HB} \rightarrow (\boxed{HB} \xrightarrow{partial} \boxed{powerset(HB)})$

The function assigns types to methods

set of methods

set of classes

types for results

Satisfaction of method signatures:

$\mathbf{I} \models c[m\text{=>}t]$ if *some* element in $I_{\text{=>}}(m)(c)$ is $\leq t$

- Basically, we want c[*m*=>t] and t::t′ to imply c[*m*=>t′]
  (if the result is of type t then it also conforms to any supertype of t)

Semantics (cont'd)

The well-typing condition:

o[*m* **->** v] is ***well-typed*** in **I**

iff whenever o $\in$ c then v $\in$ ($I_{=>}$(*m*)(c))

I is ***well-typed*** if every true object atom is well-typed.

Here we want **c**[*m* **=>** **t**], o[*m* **->** v], o**:c** to imply v**:t**.

I.e., typing is a constraint

Semantics (cont'd)

- $\mathbf{I} \models P \wedge Q$ iff $\mathbf{I} \models P$ and $\mathbf{I} \models Q$
- $\mathbf{I} \models P \vee Q$ iff $\mathbf{I} \models P$ or $\mathbf{I} \models Q$
- $\mathbf{I} \models \neg P$ iff not $\mathbf{I} \models P$
- $\mathbf{I} \models \forall ?X\ P$ iff for all $c \in HB$, $\mathbf{I} \models P'$

  P' is P with *all* free occurrences of ?X replaced with c
- $\mathbf{I} \models \exists ?X\ P$ iff for some $c \in HB$, $\mathbf{I} \models P'$

  P' is P with *some* free occurrence of ?X replaced with c

Shorthands

- $\wedge$-*Composition*: O[*m1* -> v1, *m2* -> v2] is

  O[*m1* -> v1] $\wedge$ O[*m2* -> v2]

- $\vee$-Composition: O[*m1* -> v; *m2* -> v2] is

  O[*m1* -> v1] $\vee$ O[*m2* -> v2]

- *Nesting*: O[*m1* -> v1[*m2* -> v2]] is

  O[*m1* -> v1] $\wedge$ v1[*m2* -> v2]

- IsA-Composition: O:C[*m* -> v] (or O[m ->v]:C) is

  O:C $\wedge$ O[*m* -> v]

- Same for the other arrows

These are called ***molecules*** or ***frames***

Boolean Methods

- Another shorthand: Obj[Meth]
  - E.g. ?X[p(a,?X)], f(?X)[p], john[married(1999)]
- Think of these as a shorthand for

  Obj[Meth `->` void]

  (this is only conceptually: Obj[Meth] is an independent construct and is not equivalent to Obj[Meth `->` void])
- **Boolean signatures**: Obj[=>MethType]
  - E.g., Person[=>married(Year)]

Proof Theory

- Resolution-based
  - Will see later a special case
- Sound & complete w.r.t. the semantics
  - Soundness of proofs:

    If can prove Q from a set of formulas **P** then $\mathbf{P} \models Q$
  - Completeness of proofs:

    If $\mathbf{P} \models Q$ then can prove Q from **P**

A Note on the Semantics of FLORA-2

- F-logic semantics & proof theory is completely general, like that of classical logic
- But FLORA-2 is a programming language, hence it uses non-classical semantics

  … `:-` …, `\naf` *P*, …

means: *true if cannot prove P* – so called "negation as failure."

The exact semantics for negation used in FLORA-2 is Van Gelder's Well-Founded Semantics [Van Gelder et al., JACM 1991, http://citeseer.nj.nec.com/gelder91wellfounded.html]

A Note on the Semantics (cont'd)

- The Well-Founded semantics is ***3-valued***:

  p `:- \naf` *q*.

  r `:- \naf` *r*.

  p is true, *q* false, but *r* is undefined

- And ***non-monotonic***:

  P |= Q doesn't imply P∪P′ |= Q

  p `:- \naf` *q* implies *p* true.

  But

  q and *p* `:- \naf` *q* implies *p* false.

- Classical logic is both *2-valued* and *monotonic*

Inheritance in Flora-2

- Inheritance of *structure* vs. inheritance of *behavior*
  - ***Structural inheritance*** = inheritance of the signature of a method
  - ***Behavioral inheritance*** = inheritance of the definition of a method
- Attributes/methods can be *class-level* and *object-level*
  - ***Object-level*** statements about an object, **c**, which may be a class-object, apply only to **c** and nothing else
  - ***Class-level*** statements are *inherited* from c. That is, they apply to all members of the class **c** and to all subclasses of **c**.

Structural Inheritance

- Class-level signatures appear inside class-level statements ([|…|]). Object-level signatures appear inside object-level statements ([…]).
- For ***object-level*** statements:
    - class[*method* => type] and subclass::class

      does **not** imply subclass[*method* => type]
- For ***class-level*** statements:
    - class[|*method* => type|] and subclass::class

      does imply subclass[|*method* => type|]
    - class[|*method* => type|] and obj:class

      does imply obj[*method* => type]
- *Structural inheritance is monotonic*: adding more signatures doesn't invalidate old inferences

Structural Inheritance - Semantics

Added

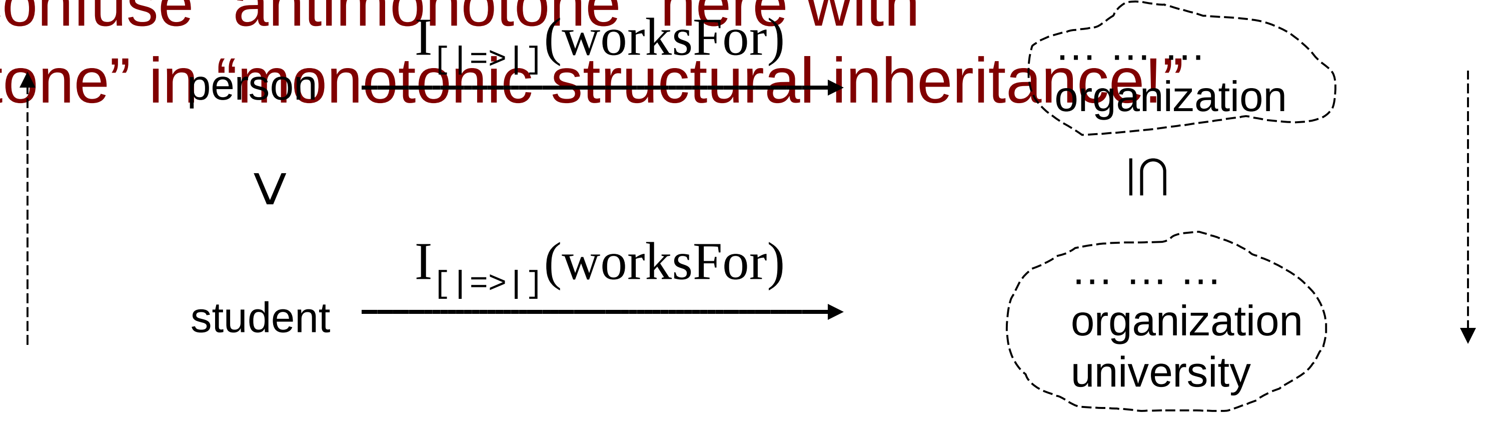
Interpretation: $\mathbf{I} = (HB, I_{->}, \in, <, I_{=>}, I_{[|=>|]})$

where

$$I_{[|=>|]}: \quad HB \rightarrow (HB \xrightarrow{\textit{partial} \text{ and } \textbf{\textit{antimonotone}}} powerset(HB))$$

Why antimonotonicity?

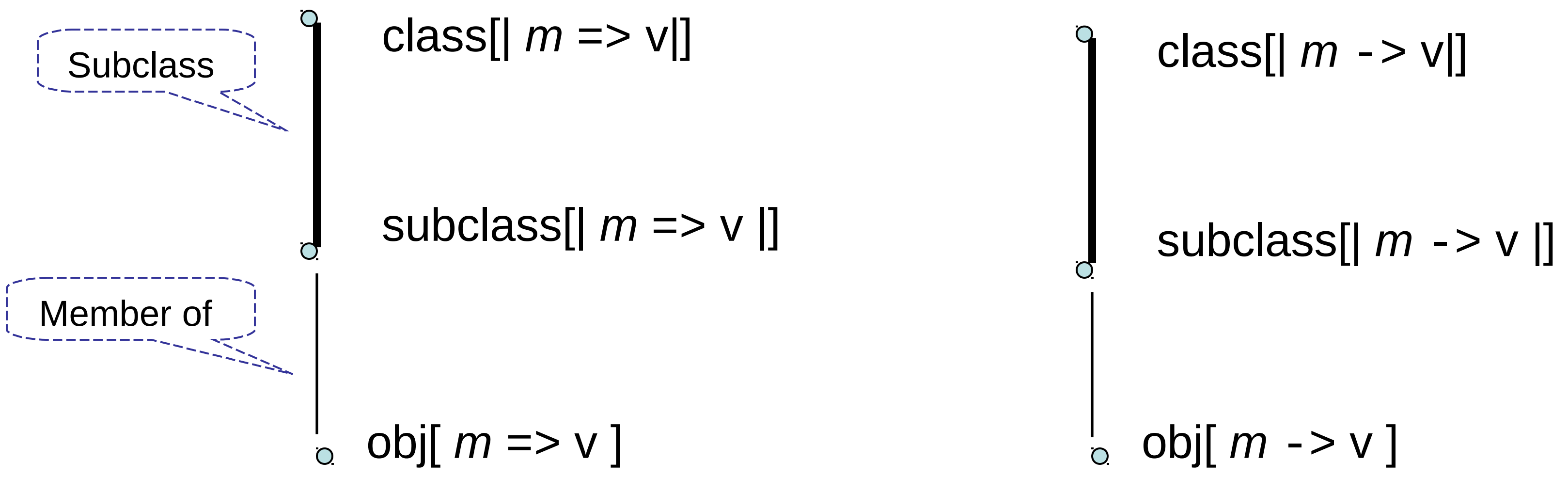
Don’t confuse “antimonotone” here with “monotone” in “monotonic structural inheritance!”

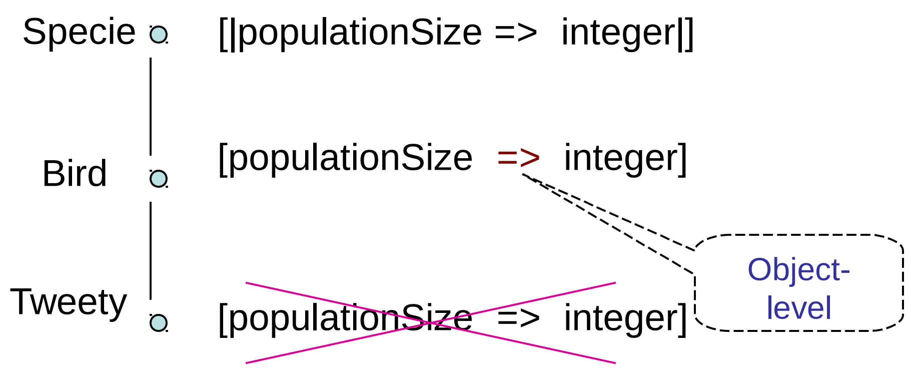


Behavioral Inheritance

- Class-level statements use `…[|…->…|]`
  - Object-level statements use `…[…->…]`
- Behavioral inheritance is *non-monotonic*

Relationship Between Inheritable and Non-inheritable Methods



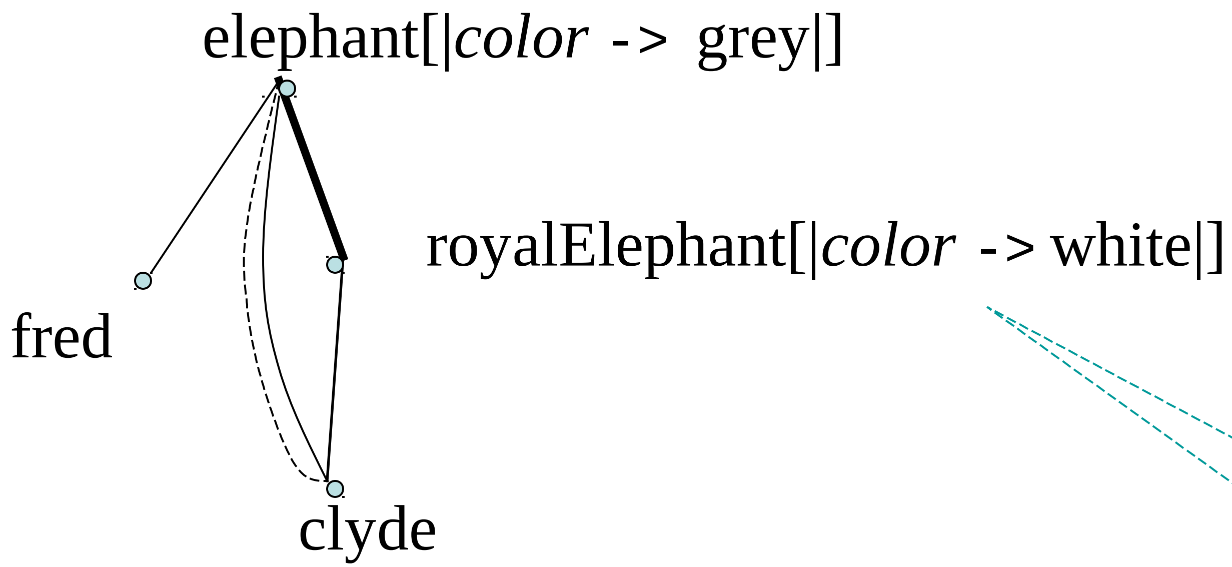


Specie
[|populationSize => integer|]
Bird
[populationSize => integer]
Tweety
~~[populationSize => integer]~~
Object-level

Inheritable methods are inherited as

- *inheritable* to subclasses
- *non-inheritable* to members

Behavioral Inheritance: Non-monotonicity


elephant[|*color* -> grey|]
royalElephant[|*color* -> white|]
fred
clyde

Overriding

Inherited:

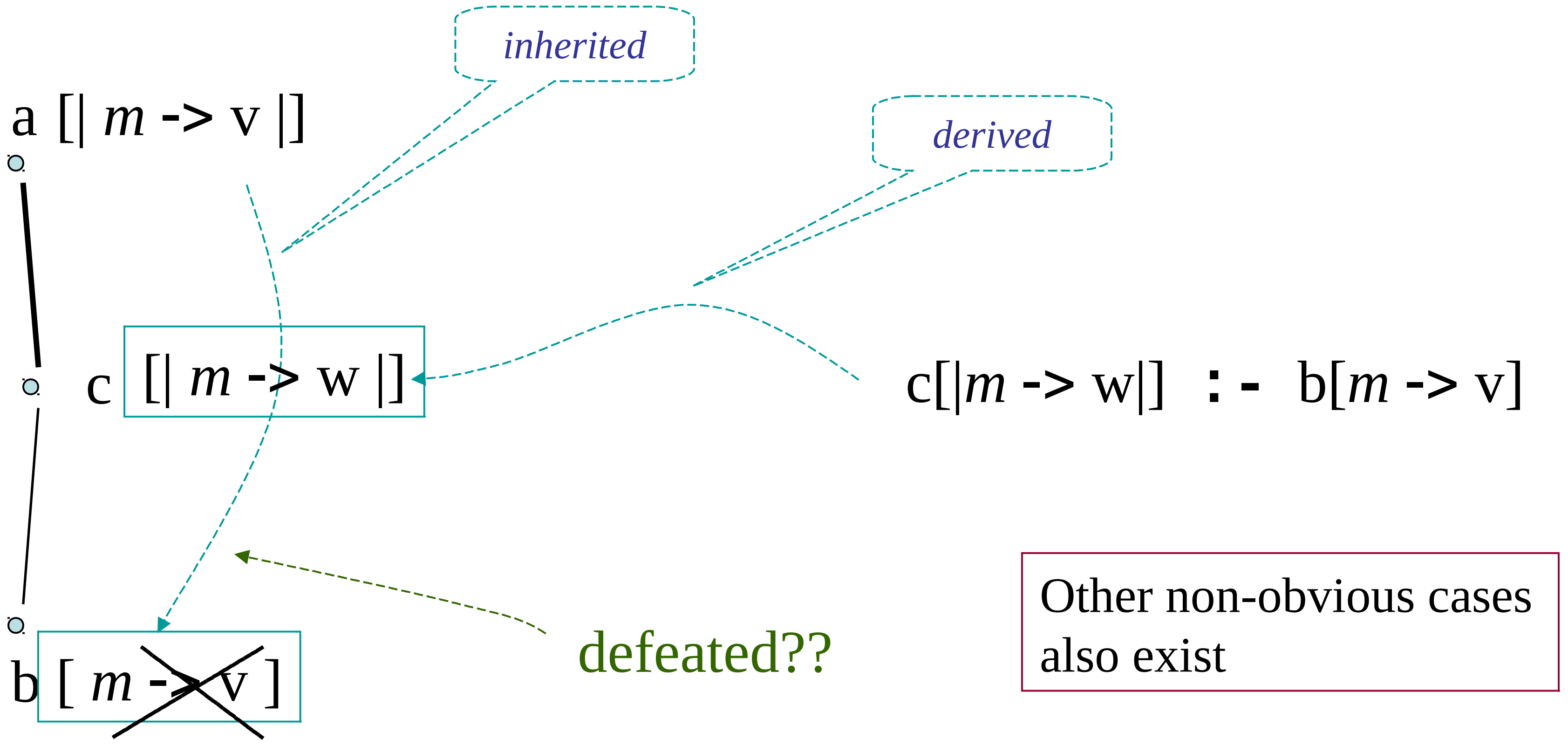
fred[*color* -> grey]

white

clyde[*color* -> ~~grey~~] ?

Behavioral Inheritance: Problem with Rules

- Inheritance is hard to even define properly in the presence of rules.



Behavioural Inheritance: Solutions

- Hard to define semantics for multiple inheritance + overriding + rules
    - Several semantics might look “reasonable”
    - Should have no unnecessary restrictions
- The original semantics in [Kifer,Lausen,Wu: JACM-95] was one of the problematic “reasonable” semantics
    - A number of other problematic semantics of various degrees of “reasonableness” exist
- Problem solved in [Yang&Kifer: Journal on Data Semantics 2006]
    - Based on semantic postulates
    - An extension of Van Gelder’s Well-Founded Semantics for negation

2.2. Background: HiLog

HiLog

- Allows certain forms of logically clean meta-programming
- Syntactically appears to be higher-order, but semantically is first-order and tractable
- Has sound and complete proof theory
- [Chen,Kifer,Warren, HiLog: A Foundation for Higher-Order Logic Programming, J. of Logic Programming, 1993]
  – The recent work on SKIF and Common Logic (Hayes et. al.) is a rediscovery of HiLog with very minor differences – 12 years later!

Examples of HiLog

Variables over predicates and function symbols:

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

Variables over atomic formulas (*reification*):

```
call(?X) :- ?X.
```

A use of HiLog in **FLORA-2** (e.g., querying of schema)**:**

```
?O[unaryMethods(?Class) -> ?M]  :-
        ?O[?M(?) ->?V],  ?V:?Class.
```

Meta-variable: ranges over method names

Syntax and Semantics of HiLog

- In predicate logic, predicates and functions are disjoint, but predicate expressions (*atomic formulas*) and functional expressions (*function terms*) have the same syntax: e.g., p(?X, f(a,b)) vs. g(?X,f(a,b))
- HiLog makes no distinction between predicates and function symbols and atomic formulas are indistinguishable from function terms

Syntax of HiLog

- Everything is built out of constant symbols and variables
- **HiLog term**:
  - ?X and f (if ?X is a variable, f – a constant)
  - F(A_1,…,A_n) if F, A_1,…,A_n are HiLog terms
  - Note: these are HiLog terms
    - Any Prolog term is, of course, a HiLog term
    - X(a,f(?Y)), f(f(f,g),?Y(?Y,?Y)), h, ?Y
    - ?X(a,f(Y))(f(f(f,g),Y(Y,Y)), h,Y)
    - ?X(a,f(?Y))(X(a,f(?Y)))(f(f(f,g),?Y(?Y,?Y)), h,?Y)

The "weird" ones

- **HiLog formula**:
  - Any HiLog term
  - A∨B, A∧B, ¬A, ∀X A, etc., if A, B are Hilog formulas

Syntax of HiLog: What are the “Weird” terms for?

- Generic transitive closure:

```
transClosure(?P)(?X,?Y) :-  ?P(?X,?Y).
transClosure(?P)(?X,?Y) :-  ?P(?X,?Z), transClosure(?P)(?Z,?Y).
```

- For instance:
  - transClosure(*parent*) is the ancestor relation
  - transClosure(*edge*)   pairs of all reachable nodes in the graph defined by *edge*

Semantics of HiLog

- **Interpretation (**Herbrand, for simplicity**):**
  - **I** = any set of variable-free HiLog terms
  - **I** $\models$ a (atomic variable-free), if a $\in$ **I**
  - **I** $\models \phi \wedge \psi$, if **I** $\models \phi$ and **I** $\models \psi$
  - etc. (as usual)
  - **I** $\models \forall X\, \phi$, if for all constant symbols c, **I** $\models \phi[X\backslash c]$, where $\phi[X\backslash c]$ is ϕ with free occurrences of X replaced with c

Relationship to Predicate Logic

- $|=_{\text{classical}}\ \psi$ implies $|=_{\text{hilog}}\ \psi$
- $|=_{\text{hilog}}\ \psi$ does ***not*** imply $|=_{\text{classical}}\ \psi$:
  - (q(a) <-> r(a)) <- $\forall X \forall Y(X=Y)$

    is valid in HiLog but not in predicate logic
- But:
  - $|=_{\text{hilog}}\ \psi$ implies $|=_{\text{classical}}\ \psi$, except for formulas that are true in every interpretation with at least γ elements in the domain (for some $\gamma > 0$), but are false in some interpretation that has less than γ elements [Chen,Kifer,Warren JLP-93].
  - Examples: Horn clauses without "=" in the head;

    Any set of "="-free formulas

Reification:

An Application of HiLog to F-logic

- **Reification**: makes an object out of a statement:

  john[*believes* -> **${**mary[*likes* -> bob]**}**]

- Introduced in [Yang & Kifer, ODBASE 2002]

Object made out of the statement mary[*likes* -> bob]

- **Main idea**:
  - Extend the syntax of F-logic to allow terms of the form

    ${**mary[*likes* -> bob]**}**, **${bob[*name* -> 'Bob Doe']**}**

    and even more general ones, like

    ${mary[*likes* -> bob, *name* -> 'Bob Doe']**}**
  - Eliminate the distinction between atomic formulas and terms both in the syntax and semantics (like in HiLog)

The Role of HiLog

- HiLog and its applications to F-logic (*reification*, *schema browsing*) allows high degree of meta-programming purely in logic
- Variables can be bound to predicate and function symbols and thus queried (e.g., which relation mentions constant ‘john’)
- Formulas can be represented as terms, decomposed, composed, and manipulated with in flexible ways
- One can mix frame syntax (F-logic) and predicate syntax (HiLog) in the same query/program:

  a[b -> c, g(?X,e) -> d], p(f(?X),a).

2.3. Background: Transaction Logic

Transaction Logic

- A logic of change
- Unlike temporal/dynamic/process logics, it is also a logic for *programming* (but can be used for *reasoning* as well)
- In the object-oriented context:
  - A logic-based language for programming the *behavior* of objects, i.e., specifying methods that change the object state

[Bonner&Kifer, An Overview of Transaction Logic, in *Theoretical Computer Science*, 1995],

[Bonner&Kifer, *A Logic for Programming Database Transactions*, in *Logics for Databases and Information Systems*, Chomicki+Saake (eds), Kluwer, 1998].

[Bonner&Kifer, Results on Reasoning about Action in Transaction Logic, in Transactions and Change in Logic Databases, *LNCS 1472*, 1998].

What's Wrong with Other Logics for Specifying Change?

- Designed for reasoning, *not* programming
    - E.g., situation calculus, temporal, dynamic, process logics
- Typically lack such basic facility as subroutines
- None became the basis for a reasonably useful programming language

Problems with Specifying Change in Logic Programming (Prolog)?

- *assert*/*retract* have no logical semantics
- Non-backtrackable, e.g.,

  ?- assert(p), q.

  If q is false, p stays.
- Prolog programs with updates are the hardest to write, debug, and understand

Example: Stacking a Pyramid

Program:

```
stack(0,X).
stack(N,X) :- N>0, move(Y,X), stack(N-1,Y).

move(X,Y) :- pickup(X), putdown(X,Y).
pickup(X)   :- clear(X), on(X,Y), retract(on(X,Y)), assert(clear(Y)).
putdown(X,Y) :- wider(Y,X), clear(Y), assert(on(X,Y)), retract(clear(Y)).
```

Action:

```
?- stack(18,block32).   // stack 18-block pyramid on top of block 32
```

Note:

Prolog *won't* execute this intuitively correct program properly!

Syntax

- Serial conjunction, $\otimes$ (often denoted using ",")
  - $a \otimes b$ – do a then do b
- The usual $\wedge$, $\vee$, $\neg$, $\forall$, $\exists$ (but with a different semantics)
  - Example: $a \vee (b \otimes \mathbf{c}) \wedge (d \vee \neg e)$
- a :- $b \equiv a \vee \neg b$
  - Means: to execute a one must execute b (i.e., a is a subroutine)
- Transaction logic also has hypothetical operators $\Diamond$ and $\Box$, but won't discuss (not implemented in FLORA-2)

Semantics

- Model-theoretic, like F-logic and HiLog
- The basic ideas
  - ***Execution path*** ≡ sequence of database states
    - Assume that the states are just sets of facts
  - Truth values over paths, not over states
  - Truth over a path ≡ ***execution*** over that path
  - ***Elementary state transitions*** ≡ propositions that cause a priori defined state transitions
    - For most purposes, can use the following elementary state transitions: t_insert{fact} and t_delete{fact} (for *transactional* insert and delete)

      t_insert{fact}: **D** $\rightarrow$ **D** + fact - add *fact* to state **D**

      t_delete{fact}: **D** $\rightarrow$ **D** – fact - delete *fact* from state **D**
    - FLORA-2 allows more powerful state transitions (**bulk updates**):

      t_insert{fact(?X)|condition(?X)} and t_delete{fact(?X)|condition(?X)}

      Insert/delete things of the form fact(X) that satisfy condition(X).

Path Structures

- Semantics is defined using the notion of path structures (which play the same role as semantic structures in classical logic)
- A ***path structure*** maps execution paths to the ordinary semantic structures used in classical predicate logic:

  I(π) = M , where π - path, M – classical semantic structure, which says which transactions can execute along the path π

  In addition:

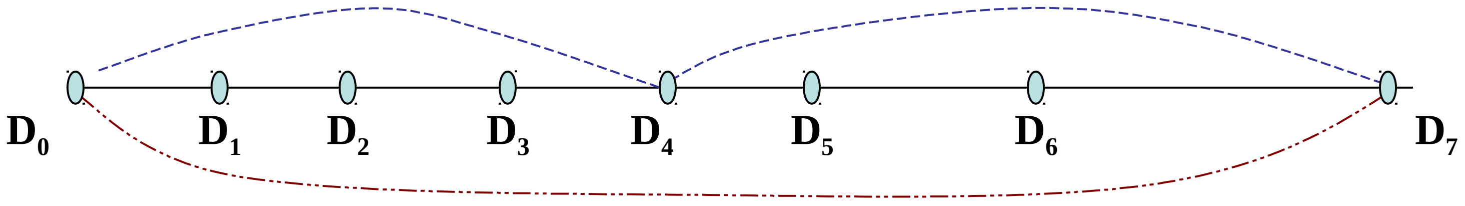
  - If π = <**D**> is a path that consists of only one database state then **I**(π) must make every fact in **D** true.
  - If π = <**D**, **D**+fact> then **I**(π) should make t_insert{fact} true
  - If π = <**D**, **D**-fact> then **I**(π) should make t_delete{fact} true

Satisfaction

Intuition:

$a \otimes b$: First execute *a* then *b* - represents sequencing of actions

If: $\mathbf{I}(<D_0,\ldots,D_4>) \mid= a$ $\mathbf{I}(<D_4,\ldots,D_7>) \mid= b$



Then: $\mathbf{I}(<D_0,\ldots,D_7>) \mid= a \otimes b$

Definition:

$\mathbf{I}(<D_0,\ldots,D_n>) \mid= a \otimes b$ iff $\exists\, D_k$ such that $\mathbf{I}(<D_0,\ldots,D_k>) \mid= a$ and $\mathbf{I}(<D_k,\ldots,D_n>) \mid= b$

Satisfaction (cont'd)

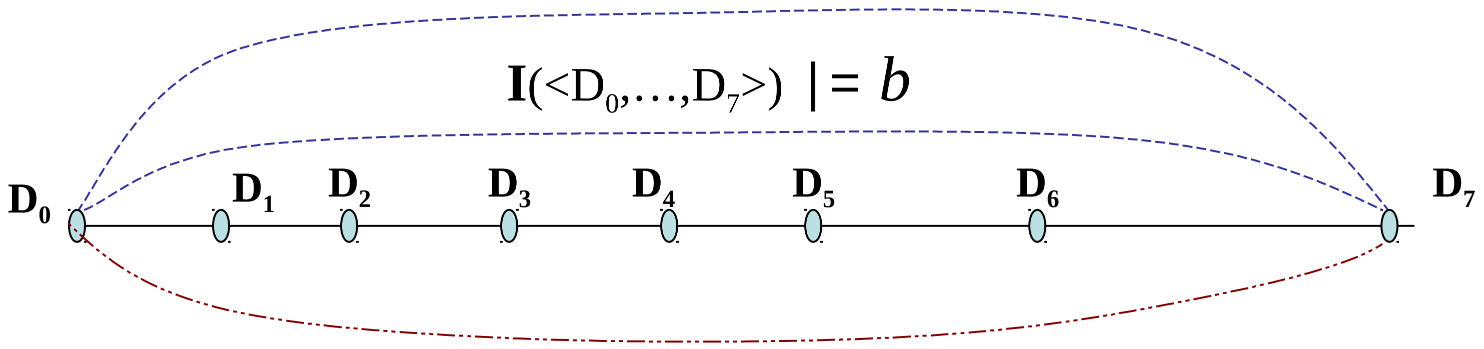
Intuition:

a /\ *b*: Execute *a* along a path that is also an execution of *b* - represents constraints

If:

$\mathbf{I}(<D_0,\ldots,D_7>) \models a$

$\mathbf{I}(<D_0,\ldots,D_7>) \models b$



Then: $\mathbf{I}(<D_0,\ldots,D_7>) \models a \wedge b$

Definition:

$\mathbf{I}(<D_0,\ldots,D_n>) \models a \wedge b$ iff $\mathbf{I}(<D_0,\ldots,D_n>) \models a$ and $\mathbf{I}(<D_0,\ldots,D_n>) \models b$

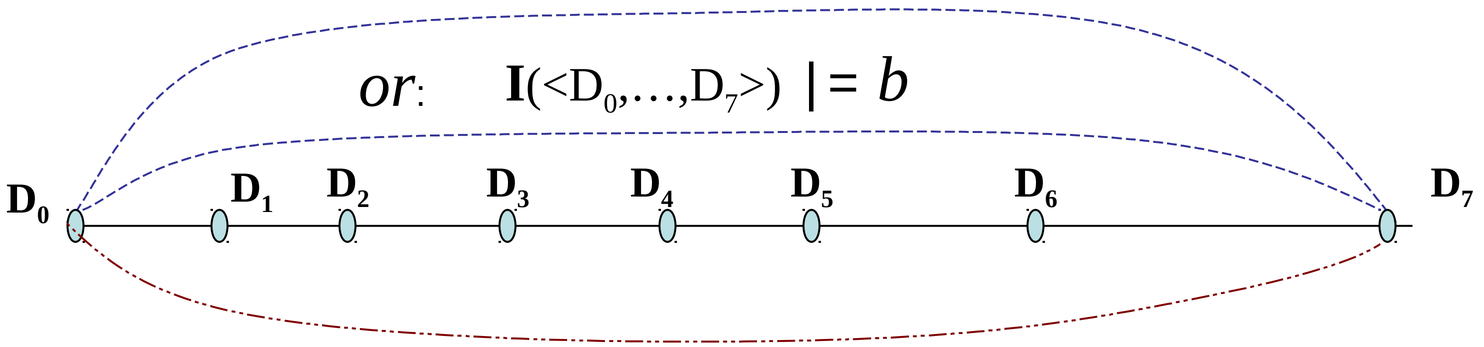
Satisfaction (cont'd)

Intuition:

$a \vee b$: Execute a along a path or execute b - represents choice

If: $\mathbf{I}(<D_0,\ldots,D_7>) \models a$

or: $\mathbf{I}(<D_0,\ldots,D_7>) \models b$



Then: $\mathbf{I}(<D_0,\ldots,D_7>) \models a \vee b$

Definition:

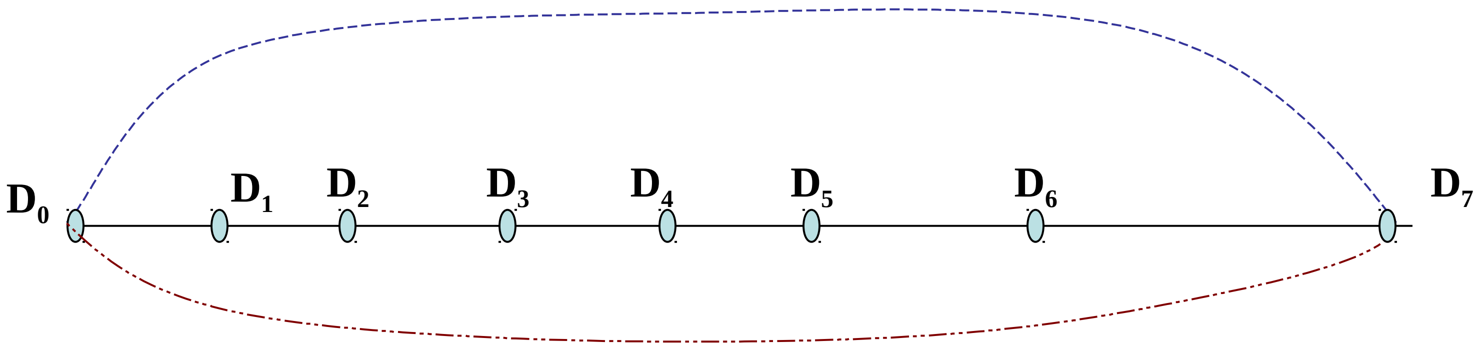
$\mathbf{I}(<D_0,\ldots,D_n>) \models a \vee b$ iff $\mathbf{I}(<D_0,\ldots,D_n>) \models a$ or $\mathbf{I}(<D_0,\ldots,D_n>) \models b$

Satisfaction (cont'd)

Intuition:

- ¬ *a*: Execute in any way provided that it is not an execution of *a*

If: $\mathbf{I}(<D_0,\ldots,D_7>) \;|\!\neq\; a$



Then: $\mathbf{I}(<D_0,\ldots,D_7>) \;|\!=\; \neg a$

Definition:

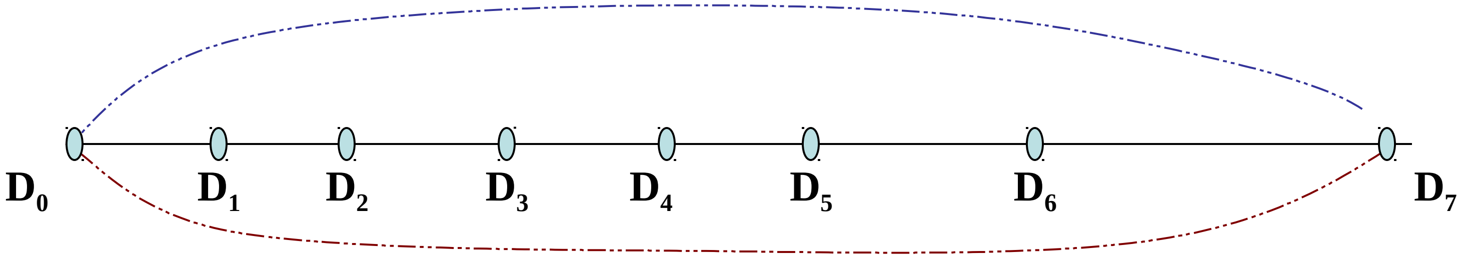
$\mathbf{I}(<D_0,\ldots,D_n>) \;|\!=\; \neg a$ iff $\mathbf{I}(<D_0,\ldots,D_n>) \;|\!\neq\; a$

Satisfaction (cont'd)

head <- *body* (defined as $a \vee \neg b$)

Formally: Every execution of body is also an execution of the head:

If: $\mathbf{I}(<D_0,\ldots,D_7>) \models body$

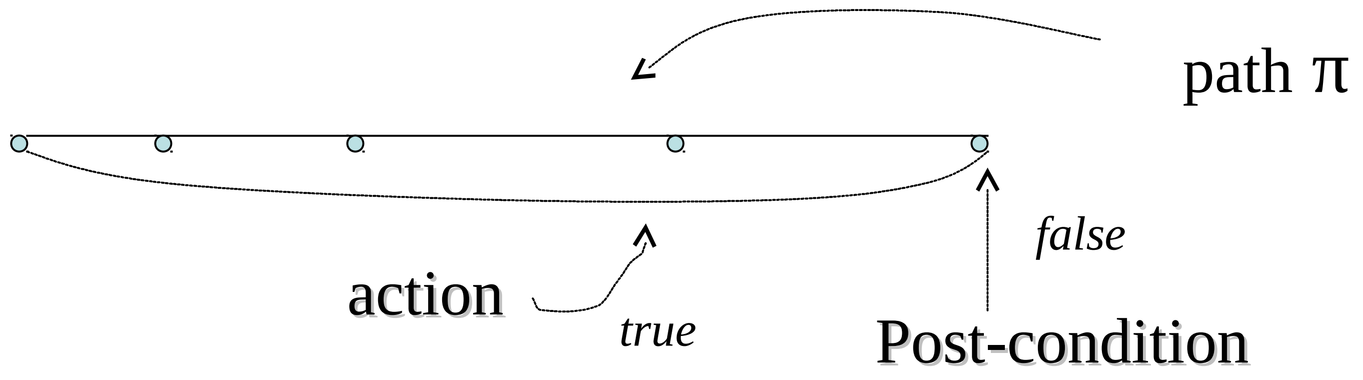


Then: $\mathbf{I}(<D_0,\ldots,D_7>) \models head$

Informally: One way to execute *head* is to execute *body*
=> *head* is the name of a procedure
and *body* is part of its definition

Properties of the Semantics

The semantics has the "*all or nothing*" flavor which makes updates logical:



If action is *true*, but postcondition *false*, then action ⊗ postcondition is *false* on π.

In practical terms: *updates are undone on backtracking*.

Transaction Programs

- A ***transaction program*** P is a set of rules of the form *head* **:-** *body* like

  move(?*X*,?*Y*) **:-** *pickup*(?*X*), *putdown*(?*X*,?*Y*)

  which define complex transactions using simple actions (like t_insert/t_delete)
- A ***transaction*** (or action) is a query of the form

  ?- *body*.

  (e.g., **?-** *stack*(*18*,*block32*))

Proof Theory

- *Executional entailment*: **P** is a set of rules, φ is a transaction (query), $D_1,\ldots,D_n$ – a sequence of states. Then

  $\mathbf{P}, D_1,\ldots,D_n \models \varphi$

  iff $\forall$ path structures **I** where $\mathbf{I} \models \mathbf{P}$ (ie., $\forall$ path π, $\mathbf{I}(\pi) \models \mathbf{P}$),

  it follows that $\mathbf{I}(<D_1,\ldots,D_n>) \models \varphi$

- To prove φ from a set of rules (transaction definitions) **P**, the proof theory tries to find a path, $D_1,\ldots,D_n$, on which φ is executionally entailed by **P**.
  - Thus, the proof theory ***executes*** φ as it proves it (and changes the underlying database state from the initial state D_1 to the final state D_n)

Pyramid Building (again)

```
stack(0,?X).
stack(?N,?X) :- ?N>0 ⊗ move(?Y,?X) ⊗ stack(?N-1,?Y).


move(?X,?Y) :- pickup(?X) ⊗ putdown(?X,?Y).
pickup(?X)   :- clear(?X) ⊗ on(?X,?Y) ⊗ t_delete{on(?X,?Y)} ⊗ t_insert{clear(?Y)}.
putdown(?X,?Y) :- wider(?Y,?X) ⊗ clear(?Y) ⊗ t_insert{on(?X,?Y)} ⊗ t_delete{clear(?Y)}.

?- stack(18,block32).          // stack 18-block pyramid on top of block 32
```

- Under the Transaction Logic semantics the above program does the right thing

Constraints

- Can express not only execution, but all kinds of sophisticated constraints:

?- *stack*(*10*, block43)
  $\wedge\ \ \forall ?X,?Y\ (move(?X,?Y) \otimes color(?X,red))\ \Rightarrow\ (\exists\ ?Z\ color(?Z,blue) \otimes move(?Z,?X))$

Whenever a red block is stacked, the next block to be stacked must be blue

- Extensions (concurrent, game-theoretic) have been shown useful for process modeling

  [Davulcu, Kifer, Ramakrishnan, & Ramakrishnan, Logic Based Modeling and Analysis of Workflows, in Proceedings of *PODS, 1997*]

  [Davulcu, Kifer, Ramakrishnan, CTR-S: A Logic for Specifying Contracts in Semantic Web Services, Proceedings of WWW2004]

Reasoning

- Can be used to *reason* about the effects of actions such as:
  - If ϕ was true before the execution of transaction then ψ must be true after
  - If ϕ was true after the execution of transaction then ψ must have been true before

[Bonner&Kifer, Results on Reasoning about Action in Transaction Logic, in *Transactions and Change in Logic Databases*, *LNCS 1472*, 1998]

Planning

- Transaction Logic is ideal for specifying planning strategies.
- **The planning problem**:
  - *Given*:
    - A set of *primitive actions* – $a_1, ..., a_n$
      each a_i can have preconditions
    - A *goal* – G
      a condition on the final state of the DB,
      which we want to achieve
    - An *initial state* D_0
  - *Find*:
    - A sequence of these actions that starting at D_0 leads to a state D that satisfies G.

Naïve Planning is Easy in Transaction Logic

Specification:

plan `:-` *action* ⊗ *plan*.

plan `:-` *action*.

action `:-` a_1.

… … …

action `:-` a_n.

To find a plan, just pose the query

`?-` *plan* ⊗ *goal*.

Example:

`?-` *plan* ⊗ (*on*(*b*,*c*)/\on(*c*,*d*)/\clear(b)).

Problem:

Proof theory might search through all sequences.

Planning with Heuristics

- Planning strategies employ heuristics to avoid exhaustive search
- Transaction Logic is ideal for specifying (and executing!) such heuristics
- Will illustrate using STRIPS (a classic planning system) as an example

STRIPS

- Uses actions of the form:

Name:	*unstack*(?*X*,?*Y*)
Comment:	Pick up block X from block Y
Precondition:	*handempty*, *clear*(?*X*), *on*(?*X*,?*Y*)
Delete:	*handempty*, *clear*(?*X*), *on*(?*X*,?*Y*)
Insert:	*clear*(?*Y*), *holding*(?*X*)

- Uses an ad hoc algorithm to construct plans
- Most AI planning systems use ad hoc algorithms
- We can write planning strategies at the high level in Transaction Logic without worrying about the low-level details

Specifying STRIPS in Transaction Logic

- First, write a rule for each action – straightforward

$$\begin{aligned} &unstack(?X,?Y) \texttt{ :- } handempty \otimes clear(?X) \otimes on(?X,?Y) \\ &\qquad \otimes \text{t\_delete}\{clear(?X), on(?X,?Y), handempty\} \\ &\qquad \otimes \text{t\_insert}\{holding(?X), clear(?Y)\} \end{aligned}$$

STRIPS in Transaction Logic (cont'd)

- Next, show how to *achieve* each goal of interest

  achieve_clear(?*Y*) :- *achieve_unstack*(?*X*,?*Y*).

  achieve_holding(?*X*) :- *achieve_unstack*(?*X*,?*Y*).

  achieve_unstack(?*X*,?*Y*) :-

  (*achieve_clear*(?*X*) * *achieve_on*(?*X*,?*Y*) * *achieve_handempty*)

  ⊗ *unstack*(?*X*,?*Y*).

  (We use $a*b$ as a shorthand for $(a \otimes b) \vee (b \otimes a)$.)

- The above says:
  - To achieve a goal, achieve the precondition of an action that inserts that goal
  - To achieve a precondition, achieve each of the subgoals in that precondition

STRIPS in Transaction Logic (cont’d)

- Base case: if a goal is already true, then it has been achieved

  achieve_on(?*X*,?*Y*) :- *on*(?*X*,?*Y*).

  achieve_clear(?*X*) :- *clear*(?*X*).

  achieve_holding(?*X*) :- *holding*(?*X*).

  achieve_handempty :- *handempty*.

STRIPS in Transaction Logic (cont'd)

- A STRIPS planning query in Transaction Logic
  - Stack c on d and b on c
    - **?-** $(achieve\_on(b,c) * achieve\_on(c,d)) \otimes on(b,c) \otimes on(c,d)$.
- The above is "ultimate" STRIPS: it finds a solution when one exists
- STRIPS was not based on a logic, so they kept refining their ad hoc execution mechanism
  - The original STRIPS was not complete. Was made complete after a series of papers
- The right logic makes the whole problem almost trivial!

Concurrent Transaction Logic

- Extends Transaction Logic with two connectives:
  - *a* | *b* – *parallel conjunction*, denotes parallel execution
  - ●*a* – *isolation*, denotes isolated execution (in the sense of transaction processing)
  - Extends the model theory and the proof theory of Transaction Logic

  [Bonner&Kifer, Concurrency and Communication in Transaction Logic, in *Joint Int'l Conference and Symposium on Logic Programming*, MIT Press, 1996]
- Suitable for process modeling and programming concurrent systems

  [Davulcu, Kifer, Ramakrishnan, & Ramakrishnan, Logic Based Modeling and Analysis of Workflows, in *Proceedings of PODS*, 1997]
- Harder to implement (not implemented in FLORA-2)
  - An interpreter available at http://www.cs.toronto.edu/~bonner/ctr/

Concurrent Transaction Logic for Services

- Extends Concurrent Transaction Logic with one additional connective:

  $a \prod b$ – the *opponent's conjunction*
- Enables specification of the *behavioral aspects* of *service contracts*
  - When different parties to the contract can make different choices (e.g., ship insured or uninsured, pay in full or in installments)

- [Davulcu, Kifer, & Ramakrishnan, CTR-S: A Logic for Specifying Contracts in Semantic Web Services, *WWW 2004*, May 2004]

2.4. Background: Top-down Execution and Tabling

SLD-Resolution

- Strategy at the core of any top-down execution engine
- *Sound* inference strategy
- *Complete* only for pure ***Horn*** clauses, i.e.,
  - Set of ***rules***: *head* **:-** *body* where *head* is atomic (of the form p(…)) and *body* is $b_1, \ldots, b_n$ (conjunction of atomic formulas). No negation in the head or the rule body.
    - Can be viewed as head $\lor \neg b_1 \lor \ldots \lor \neg b_n$
  - Set of ***facts***: atomic formulas.
    - Same syntax as *head*.
    - Can be viewed as a rule with empty body.
  - ***Goal***: same syntax as the rule body.
    - The purpose of SLD resolution is to prove that $\exists ?X$ *goal* (?*X* represents all the vars in *goal*) follows from the set of facts plus the set of rules
    - Find all *x* such that *goal*[?*X*\x] (goal in which all occurrences of ?*X* are replaced with *x*) is implied by *rules* + *facts*.

SLD (cont'd)

- Goal: $g_1,\ldots,g_k$

  Rule: h **:-** $b_1,\ldots,b_n$

  ☝Rename vars in the rule to be disjoint from the vars in goal

  θ: most general substitution s.t. $h\theta = g_1\theta$
- Derive new goal: $(b_1,\ldots,b_n, g_2,\ldots,g_k)\theta$

  Note: g_1 replaced with $b_1,\ldots,b_n$
- *Example*:
  - Goal: p(?X,f(?Y)), q(?X,?Y,?Z)
  - Rule: p(g(?V),?W) **:-** r(?V,f(?W)), h(?W,?U).
  - θ: ?X -> g(?V), ?W -> f(?Y)
  - Derived goal: r(?V,f(f(?Y))), h(f(?Y),?U), q(g(?V),?Y,?Z)

SLG (SLD with negation)

- When rules have negation in the body, the logically sound approach is to use the 3-valued Well-Founded Semantics (mentioned earlier)
- The adaptation of SLD to this case is called ***SLG Resolution***. [Swift and Warren, *Intl. Logic Programming Symposium*, 1994]
  - *Roughly* works as SLD, but when it sees `\naf` *p* in the rule body, tries to prove *p*, possibly delaying until the literals to the right of `\naf` *p* have been proved. Three outcomes:
    - Proved *p*: `\naf` *p* is false
    - Proved that *p* cannot be proved: `\naf` *p* is true
    - All ways of deriving *p* rely on assuming `\naf` *p*: *p* is undefined

Prolog Execution Strategy

- What if several rules have heads that unify with g_1 in $g_1,\ldots,g_k$?
  - SLD doesn't assume any order in which these rules are tried. If all orders are tried, then SLD is complete for Horn rules
  - Prolog does assume an order: rules are tried in the order in which they occur in the program. This causes Prolog to miss solutions even if they exist:

```
Goal:   ?-  p(?X)
Rules:  p(?X) :-   p(?X).
        p(?X) :-  r(?X).
        r(a).
```

    - Prolog will get stuck in an infinite loop due to the first rule

Solution: Tabling

- When an attempt to solve a literal in the rule body is made (a ***call*** to the literal is made), save it in a table
- If the same call is made again, don't use SLD – look up the table instead; feed the answers from the first call to the second. Meanwhile, explore the other possibilities
- Example:

```
Goal:  ?-  p(?X)
Rules: p(?X) :-  p(?X).
       p(?X) :-  r(?X).
       r(a).
```

Call to p(?X). Save it in the table.
First derivation branch:
Use SLD with rule #1;
- create another call to p(?X).
- Look up the table—don't execute!
- Postpone this derivation branch.

Second derivation branch: Use SLD with rule #2
Call to r(?X). Save in the table.
Resolve with the fact r(a), get a result: ?X=a
No answers in the 1st derivation branch

Tabling (cont'd)

- See [Warren, CACM 1992]
- SLG resolution incorporates tabling
- SLG (unlike Prolog) is complete for Horn clauses; it is complete for the Well-Founded semantics for queries with negation in the rule body
- XSB is the only complete implementation of SLG
- YAP (http://yap.sourceforge.net) has an implementation of tabling; aims at having a complete implementation in the future

SLD and SLG in F-logic

- Similar to Prolog. Difference: goals and rule heads can have F-logic molecules in them:

```
Goal:    ?-  a[b -> c, d -> e].
Rules:   ?Z[b -> ?Y, f -> ?Z]  :-  body.
         ?X[d -> ?Y, h -> ?Z]  :-  anotherBody.
```

  Can these rules resolve with the goal?

- Answer: The notion of SLD resolution needs a slight modification.

SLD in F-logic (cont'd)

- Goals are transformed to eliminate disjunction (remember: disjunction is allowed in rule bodies and goals, but not in rule heads):

  ?- ?X[disj1 **;** disj2], *rest*.

  becomes a pair of goals:

  ?- ?X[disj1], *rest*.

  ?- ?X[disj2], *rest*.

  Must solve each goal and *union* the solutions.

- Note: a similar transformation is done in regular logic programming:

  ?- (p **;** q), *rest*.

  becomes

  ?- p, *rest*.

  ?- q, *rest*.

SLD in F-logic (cont'd)

- Goals are further transformed to simplify molecules:

  ?- ?X[part1 , part2], *rest*.

  becomes

  ?- ?X[part1], ?X[part2], *rest*.

  and

  ?- ?X[foo -> {bar1, bar2}], *rest*.

  becomes

  ?- ?X[foo -> bar1], ?X[foo -> bar2}], *rest*.

  Break molecules down into *atomic* (indivisible) ones.

SLD in F-logic (cont'd)

- SLD rule:

  Goal: **?-** subgoal-atomic-molecule, *rest*.

  Rule: head-molecule **:-** *body*.

  ☝ Rename vars in the rule to be disjoint from the vars in the goal

  θ: most general unifier of subgoal-atomic-molecule *into* head-molecule, i.e, θ(subgoal-atomic-molecule) ⊆ θ(head-molecule)

  (⊆ means both have the same object-term and the single component of subgoal-atomic-molecule inside the […] is one of the components of head-molecule)

  New goal: **?-** θ(*body*), θ(*rest*).

SLD in F-logic (cont'd)

- Example:
  - **?-** f(?X,a)[m1 -> ?X, m2(?Y) -> b], p(?Y).
  - ?V[?W -> c, m2(?V) -> b, m1 -> ?W] **:-** a[?V ->?W].
- – Transform:
  - **?-** f(?X,a)[m1 -> ?X], f(?X,a)[m2(?Y) -> b], p(?Y).
- – One unifier and new goal:

  θ: ?V -> f(?X,a), ?W -> m1, ?X -> c

  ?- a[f(?X,a) -> m1], f(?X,a)[m2(?Y) -> b], p(f(?X,a)).
- – Another possibility:

  θ: ?V -> f(?X,a), ?W -> ?X

  ?- a[f(?X,a) -> ?X], f(?X,a)[m2(?Y) -> b], p(f(?X,a)).

SLG in F-logic

- FLORA-2 uses Prolog-like execution strategy
  - To be complete, it uses tabling
  - For negation in the rule body, it uses the Well-Founded Semantics and thus the SLG resolution
- To support inheritance, it uses an *extended* Well-Founded semantics, as mentioned earlier.
  - This is implemented by a translation into a Prolog program, which utilizes SLG resolution