

# From Syntax to Schema & Semantics

- After *pattern-based cleaning*
  - regular expressions, OpenRefine, ...
- ... load data into a database system!
- ... and then exploit database technology:
  - *queries & integrity constraints!*
- **Relational Databases**
  - *Logic-based* approach first: **Datalog**
    - Facts, rules, queries, integrity constraints
    - Rich body of research; theory & practice!
  - *Relational data everywhere*: **SQL**



# Semantics / ICs in Conceptual Models (ER, UML, OO, ...)

- What we *can* express:
  - e.g., functional dependencies (key constraints),
  - inclusion dependencies (foreign key constraints),
  - cardinality constraints,
  - hierarchical constraints (class hierarchy), ...
- What we *cannot* express:
  - e.g., general arithmetic constraints,
  - application-specific constraints and knowledge,
  - purpose, goals, other general requirements, ...

# Dirty/Clean data as In-/Consistent data

<i>Students</i>	<i>StuNum</i>	<i>StuName</i>
	101	<i>john bell</i>
	102	<i>mary stein</i>
	104	<i>claire stevens</i>
	107	<i>pat norton</i>

<i>Enrollment</i>	<i>StuNum</i>	<i>Course</i>
	104	<i>comp150</i>
	101	<i>comp100</i>
	101	<i>comp200</i>
	105	<i>comp120</i>

Figure 1.1: A database instance.

Source: [bertossi2011database]

- Important **Integrity Constraints (ICs)**
  - **Functional Dependencies FD** ( $\approx$  key constraints)
    - one or more columns provide a unique key
  - **Inclusion Dependencies ID** ( $\approx$  referential ICs)
    - $\text{Child\_Table.FK} \rightarrow \text{Parent\_Table.PK}$

# Is this consistent?

<i>Students</i>	<i>StuNum</i>	<i>StuName</i>
	101	<i>john bell</i>
	101	<i>joe logan</i>
	104	<i>claire stevens</i>
	107	<i>pat norton</i>

<i>Enrollment</i>	<i>StuNum</i>	<i>Course</i>
	104	<i>comp150</i>
	101	<i>comp100</i>
	101	<i>comp200</i>

**Figure 1.2:** An inconsistent instance.

<i>Students1</i>	<i>StuNum</i>	<i>StuName</i>
	101	<i>john bell</i>
	104	<i>claire stevens</i>
	107	<i>pat norton</i>

<i>Students2</i>	<i>StuNum</i>	<i>StuName</i>
	101	<i>joe logan</i>
	104	<i>claire stevens</i>
	107	<i>pat norton</i>

**Figure 1.3:** Two repairs.

Source: [bertossi**2011**database]

# Integrity Constraints as Logic Rules & Queries

The two particular kinds of integrity constraints presented above, and also other forms of ICs, can be easily expressed in the language of first-order (FO) predicate logic [Enderton, 2001], which in relational databases usually takes the form of the *relational calculus* [Abiteboul et al., 1995]. For example, the FD above can be expressed by the symbolic sentence

$$\forall x \forall y \forall z ((\text{Students}(x, y) \wedge \text{Students}(x, z)) \rightarrow y = z), \quad (1.1)$$

whereas the referential constraint above can be expressed by

$$\forall x \forall y (\text{Enrollment}(x, y) \rightarrow \exists z \text{Students}(x, z)). \quad (1.2)$$

Notice that this language of FO predicate logic is determined by the database schema, whose predicates are now being used to write down logical formulas. We may also use “built-in” predicates,

Source: [bertossi2013generic]

ic\_violated(X) :- student(X,Y), student(X,Z), Y != Z.

# First-Order Queries + Datalog: 4-in-1

- SQL  $\text{SELECT} \dots \text{FROM} \dots \text{WHERE} \dots$
- Relational Algebra (RA)  $\sigma, \pi, \bowtie, \delta, \cup, \setminus$
- Relational Calculus (RC)  $\forall x F, \exists x F, F \wedge G, F \vee G, \neg F$
- **Datalog**  $\approx \text{RC} + \text{Recursion}$

EXAMPLE: Given relations  $\boxed{\text{employee(Emp, Salary, DeptNo)}}$  and  $\boxed{\text{dept(DeptNo, Mgr)}}$ , find all (employee, manager) pairs:

- SQL: 

```
SELECT Emp, Mgr
      FROM employee, dept
      WHERE employee.DeptNo = dept.DeptNo
```
- RA:  $\pi_{\text{Emp}, \text{Mgr}}(\text{employee} \bowtie \text{dept})$
- RC:  $F(\text{Emp}, \text{Mgr}) =$   
 $\exists \text{Salary, DeptNo} : (\text{employee}(\text{Emp}, \text{Salary}, \text{DeptNo}) \wedge \text{dept}(\text{DeptNo}, \text{Mgr}))$
- **Datalog**:  $\text{boss}(\text{Emp}, \text{Mgr}) \leftarrow \text{employee}(\text{Emp}, \text{Salary}, \text{DeptNo}), \text{dept}(\text{DeptNo}, \text{Mgr})$

*Two more ways to write the same query:* 

$\text{boss}(\text{E}, \text{M}) :- \text{employee}(\text{E}, \text{S}, \text{D1}), \text{dept}(\text{D2}, \text{M}), \text{D1} = \text{D2}.$

$\text{boss}(\text{E}, \text{M}) :- \text{employee}(\text{E}, \_, \text{D}), \text{dept}(\text{D}, \text{M}).$

# ASP/DLV Demo (inside Emacs)

The screenshot shows the pq.dlv Emacs interface. The top menu bar includes New, Open, Recent, Save, Print, Undo, Redo, Cut, Copy, Paste, Search, Preferences, and Help. The window title is pq.dlv. The buffer contains the following Prolog code:

```
% emp(EmpId, Salary, DeptId)
emp(e1,100, d1).
emp(e2, 80, d1). % e2 works in d1
emp(e2, 80, d2).

% dept(DeptId, ManagerID)
dept(d1, e2). % ... but e2 is also the manager of d1!
dept(d2, e3).

% IC: Every manager must also show up in the emp/3 table!
in_emp(E) :- emp(E,S,D).
ic_missing_emp(M) :- dept(_,M), not in_emp(M).

% boss(EmpId, MgrId)
boss(E,M) :- emp(E,_,D1), dept(D2,M), D1=D2.
```

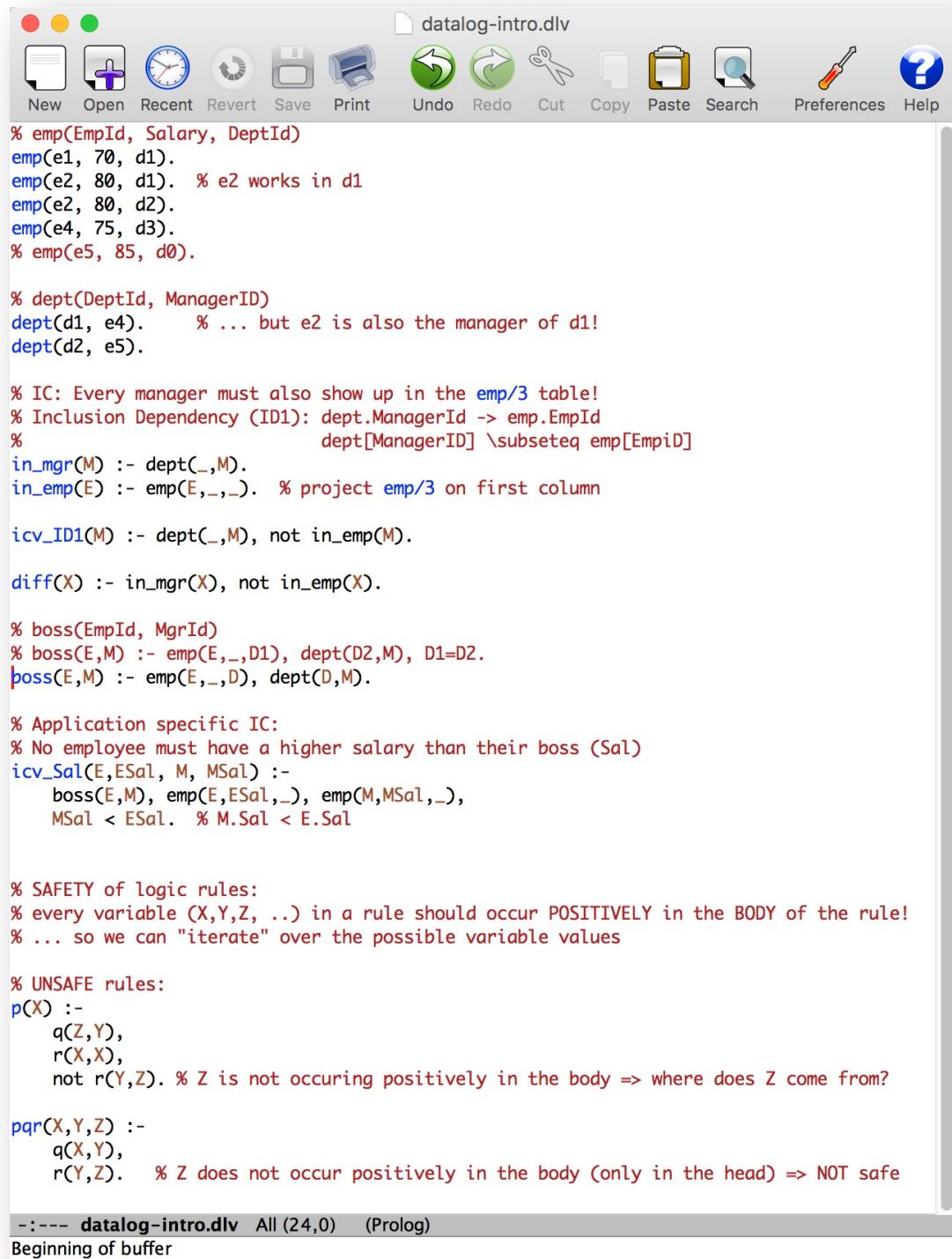
The bottom part of the interface shows the DLV shell:

```
U--- pq.dlv Top (5,0) (Prolog)
pq.dlv 1 *shell*
DLV [build BEN/Dec 21 2011 gcc 4.2.1 (Apple Inc. build 5666) (dot 3)]
{boss(e1,e2), boss(e2,e3)}
[HW2 :]$ dlv pq.dlv -filter=boss
DLV [build BEN/Dec 21 2011 gcc 4.2.1 (Apple Inc. build 5666) (dot 3)]
{boss(e1,e2), boss(e2,e2), boss(e2,e3)}
[HW2 :]$ dlv pq.dlv -filter=ic_missing_emp
DLV [build BEN/Dec 21 2011 gcc 4.2.1 (Apple Inc. build 5666) (dot 3)]
pq.dlv: line 11: Rule is not safe.
Aborting due to parser errors.
[HW2 :]$ dlv pq.dlv -filter=ic_missing_emp
DLV [build BEN/Dec 21 2011 gcc 4.2.1 (Apple Inc. build 5666) (dot 3)]
{ic_missing_emp(e3)}
[HW2 :]$
```

Two red arrows point from the text "A query: list employees with their ‘boss(es)’, i.e., the managers of the departments the employee works for." to the line "boss(E,M) :- emp(E,\_,D1), dept(D2,M), D1=D2." and the line "ic\_missing\_emp(M) :- dept(\_,M), not in\_emp(M).".

- An **integrity constraint** (IC): every dept manager should be in the emp/3 table (= foreign key constraint)
- A **query**: list employees with their “boss(es)”, i.e., the managers of the departments the employee works for.

# Integrity Constraints in Datalog



The screenshot shows a window titled "datalog-intro.dlv" with a menu bar containing "File", "Edit", "Tools", "Help", and a toolbar with icons for New, Open, Recent, Revert, Save, Print, Undo, Redo, Cut, Copy, Paste, Search, Preferences, and Help.

```
% emp(EmpId, Salary, DeptId)
emp(e1, 70, d1).
emp(e2, 80, d1). % e2 works in d1
emp(e2, 80, d2).
emp(e4, 75, d3).
% emp(e5, 85, d0).

% dept(DeptId, ManagerID)
dept(d1, e4). % ... but e2 is also the manager of d1!
dept(d2, e5).

% IC: Every manager must also show up in the emp/3 table!
% Inclusion Dependency (ID1): dept.ManagerId -> emp.EmpId
%                                     dept[ManagerID] \subseteq emp[EmpID]
in_mgr(M) :- dept(_,M).
in_emp(E) :- emp(E,_,_). % project emp/3 on first column

icv_ID1(M) :- dept(_,M), not in_emp(M).

diff(X) :- in_mgr(X), not in_emp(X).

% boss(EmpId, MgrId)
% boss(E,M) :- emp(E,_,D1), dept(D2,M), D1=D2.
boss(E,M) :- emp(E,_,D), dept(D,M).

% Application specific IC:
% No employee must have a higher salary than their boss (Sal)
icv_Sal(E,ESal, M, MSal) :-
    boss(E,M), emp(E,ESal,_), emp(M,MSal,_),
    MSal < ESal. % M.Sal < E.Sal

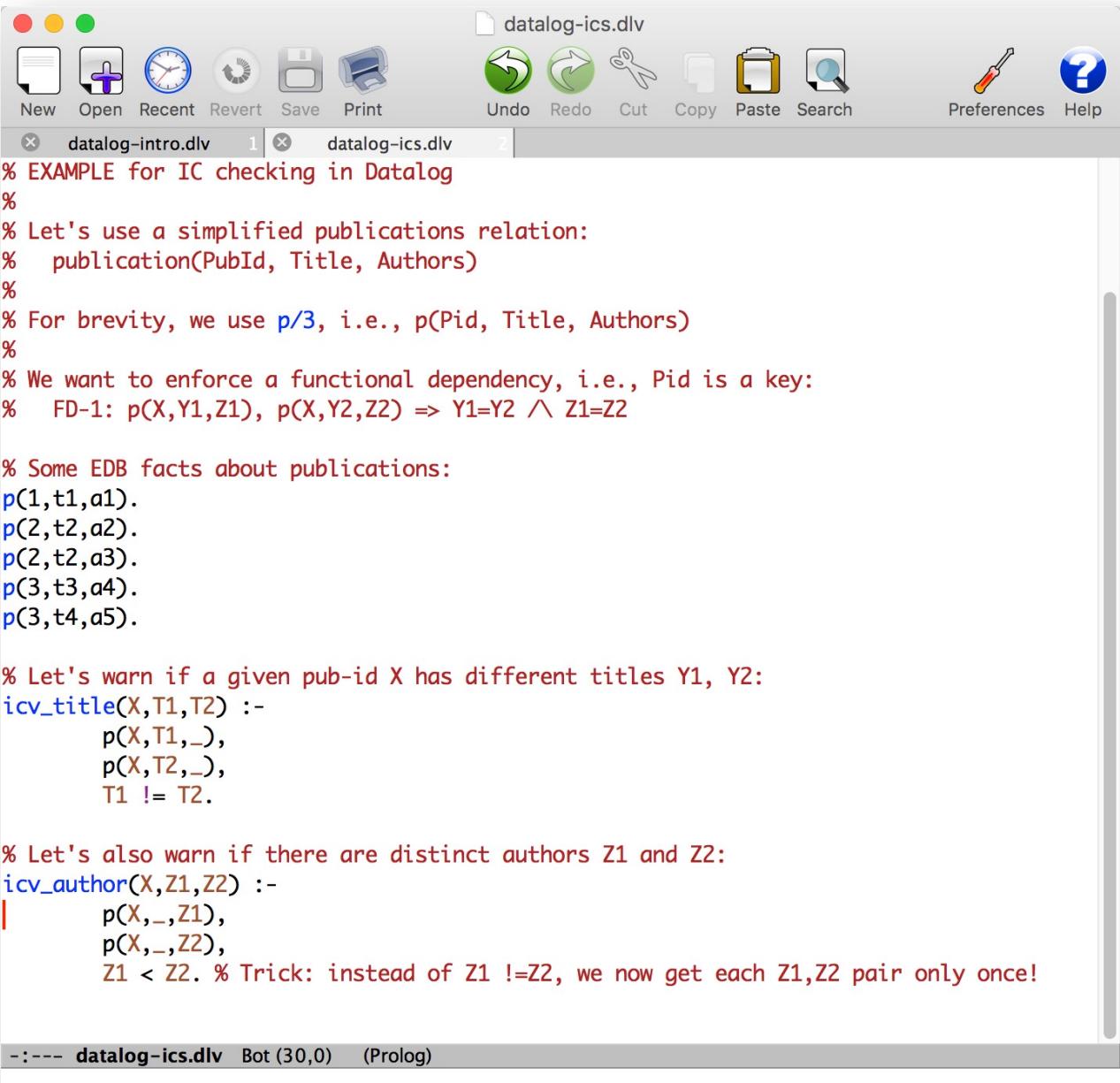
% SAFETY of logic rules:
% every variable (X,Y,Z, ...) in a rule should occur POSITIVELY in the BODY of the rule!
% ... so we can "iterate" over the possible variable values

% UNSAFE rules:
p(X) :-
    q(Z,Y),
    r(X,X),
    not r(Y,Z). % Z is not occurring positively in the body => where does Z come from?

pqr(X,Y,Z) :-
    q(X,Y),
    r(Y,Z). % Z does not occur positively in the body (only in the head) => NOT safe
```

-:--- **datalog-intro.dlv** All (24,0) (Prolog)  
Beginning of buffer

# Integrity Constraints in Datalog



```
% EXAMPLE for IC checking in Datalog
%
% Let's use a simplified publications relation:
% publication(PubId, Title, Authors)
%
% For brevity, we use p/3, i.e., p(Pid, Title, Authors)
%
% We want to enforce a functional dependency, i.e., Pid is a key:
% FD-1: p(X,Y1,Z1), p(X,Y2,Z2) => Y1=Y2 ∧ Z1=Z2

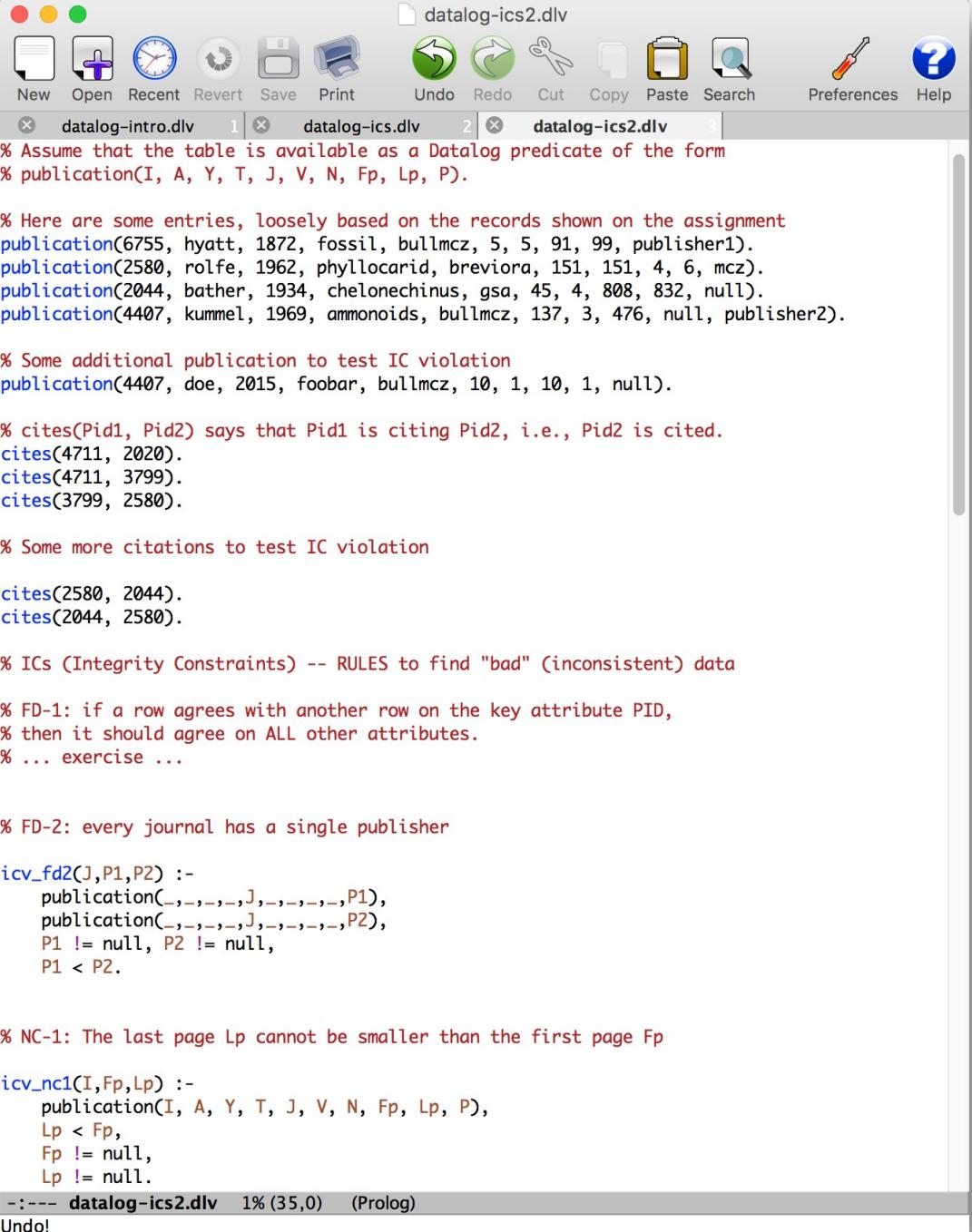
% Some EDB facts about publications:
p(1,t1,a1).
p(2,t2,a2).
p(2,t2,a3).
p(3,t3,a4).
p(3,t4,a5).

% Let's warn if a given pub-id X has different titles Y1, Y2:
icv_title(X,T1,T2) :-
    p(X,T1,_),
    p(X,T2,_),
    T1 != T2.

% Let's also warn if there are distinct authors Z1 and Z2:
icv_author(X,Z1,Z2) :-
    p(X,_,Z1),
    p(X,_,Z2),
    Z1 < Z2. % Trick: instead of Z1 != Z2, we now get each Z1,Z2 pair only once!

-:--- datalog-ics.dlv  Bot (30,0)  (Prolog)
```

# ICs in Datalog => Assignment



The screenshot shows a Datalog IDE interface with a menu bar and toolbars at the top, and a main window displaying a Prolog script. The menu bar includes 'File', 'Edit', 'Recent', 'Revert', 'Save', 'Print', 'Undo', 'Redo', 'Cut', 'Copy', 'Paste', 'Search', 'Preferences', and 'Help'. The toolbar includes icons for New, Open, Save, Print, Undo, Redo, Cut, Copy, Paste, and Search. The main window has three tabs: 'datalog-intro.dlv' (1), 'datalog-ics.dlv' (2), and 'datalog-ics2.dlv' (3). The 'datalog-ics2.dlv' tab is active, showing the following Prolog code:

```
% Assume that the table is available as a Datalog predicate of the form
% publication(I, A, Y, T, J, V, N, Fp, Lp, P).

% Here are some entries, loosely based on the records shown on the assignment
publication(6755, hyatt, 1872, fossil, bullmcz, 5, 5, 91, 99, publisher1).
publication(2580, rolfe, 1962, phyllocarid, breviora, 151, 151, 4, 6, mcz).
publication(2044, bather, 1934, chelonechinus, gsa, 45, 4, 808, 832, null).
publication(4407, kummel, 1969, ammonoids, bullmcz, 137, 3, 476, null, publisher2).

% Some additional publication to test IC violation
publication(4407, doe, 2015, foobar, bullmcz, 10, 1, 10, 1, null).

% cites(Pid1, Pid2) says that Pid1 is citing Pid2, i.e., Pid2 is cited.
cites(4711, 2020).
cites(4711, 3799).
cites(3799, 2580).

% Some more citations to test IC violation
cites(2580, 2044).
cites(2044, 2580).

% ICs (Integrity Constraints) -- RULES to find "bad" (inconsistent) data

% FD-1: if a row agrees with another row on the key attribute PID,
% then it should agree on ALL other attributes.
% ... exercise ...

% FD-2: every journal has a single publisher

icv_fd2(J,P1,P2) :-
    publication(_,_,_,_,J,_,_,_,_,P1),
    publication(_,_,_,_,J,_,_,_,_,P2),
    P1 != null, P2 != null,
    P1 < P2.

% NC-1: The last page Lp cannot be smaller than the first page Fp

icv_nc1(I,Fp,Lp) :-
    publication(I, A, Y, T, J, V, N, Fp, Lp, P),
    Lp < Fp,
    Fp != null,
    Lp != null.

:- --- datalog-ics2.dlv 1% (35,0) (Prolog)
```

Undo!

# From IC Checking to IC Repair ..

184

L. Bertossi and L. Bravo

**Fig. 1** A database instance

<i>Students</i>		<i>Enrollment</i>	
<i>StuNum</i>	<i>StuName</i>	<i>StuNum</i>	<i>Course</i>
101	john bell	104	comp150
102	mary stein	101	comp100
104	claire stevens	101	comp200
107	pat norton	105	comp120

**Fig. 2** Another instance

<i>Students</i>		<i>Enrollment</i>	
<i>StuNum</i>	<i>StuName</i>	<i>StuNum</i>	<i>Course</i>
101	john bell	104	comp150
101	joe logan	101	comp100
104	claire stevens	101	comp200
107	pat norton		

one student name. This condition, called a *functional dependency* (FD), is denoted with  $StuNumber \rightarrow StuName$ , or  $Students : StuNumber \rightarrow StuName$ , to indicate that this dependency should hold for attributes of relation *Students*. Actually, in this case, since all the attributes in the relation functionally depend on *StuNum*, the FD is called a *key constraint*.

Integrity constraints can be declared together with the schema, indicating that the instances for the schema should all satisfy the integrity constraints. For example, if the functional dependency  $Students : StuNumber \rightarrow StuName$  is added to the schema, the instance in Fig. 1 is consistent, because it satisfies the FD. However, the instance in Fig. 2 is *inconsistent*. This is because this instance does not satisfy, or, what is the same, violates the functional dependency (the student number 101 is assigned to two different student names).

# From IC Checking to IC Repair ..

It is also possible to consider with the schema a *referential integrity constraint* that requires that every student (number) in the relation *Enrollment* appears, associated with a student name, in relation *Students*, the official “table” of students. This is denoted with  $\text{Enrollment}[\text{StuNum}] \subseteq \text{Students}[\text{StuNum}]$  and is a form of *inclusion dependency*. If this IC is considered in the schema, the instance in Fig. 1 is inconsistent, because student 105 does not appear in relation *Students*. However, if only this referential constraint were associated to the schema, the instance in Fig. 2 would be consistent. The combination of the given referential constraint and the functional dependency creates a *foreign key constraint*: The values for attribute *StuNum* in *Enrollment* must appear in relation *Students* as values for its attribute *StuNum*, and this attribute form a *key* for *Students*.

It can be seen that the notion of consistency is relative to a set of integrity constraints. A database instance that satisfies each of the constraints in this set is said to be *consistent* and *inconsistent* otherwise.

The two particular kinds of integrity constraints presented above and also other forms of ICs can be easily expressed in the language of predicate logic. For example, the FD above can be expressed by the symbolic sentence

$$\forall x \forall y \forall z ((\text{Students}(x, y) \wedge \text{Students}(x, z)) \rightarrow y = z), \quad (1)$$

# From IC Checking to IC Repair ..

**Fig. 3** Two repairs of *Students* in Fig. 2

Students		Students	
<i>StuNum</i>	<i>StuName</i>	<i>StuNum</i>	<i>StuName</i>
101	john bell	101	joe logan
104	claire stevens	104	claire stevens
107	pat norton	107	pat norton

## 3 Repairs and Consistent Answers

The notion of consistency of a database is a holistic notion that applies to the entire database, and not to portions of it. In consequence, in order to pursue this idea of retrieving consistent query answers, it becomes necessary to characterize the consistent data in an inconsistent database first. The idea that was proposed in [2] is as follows: the consistent data in an inconsistent data are the data that are invariant under all possible way of restoring the consistency by performing minimal changes on the initial database. That is, no matter what minimal consistency restoration process is applied to the database, the consistent data stay in the database. Each of the consistent versions of the original instance obtained by minimal changes is called a *minimal repair*, or, simply, a *repair*.

It becomes necessary to be more precise about the meaning of minimal change. In between, a few notions have been proposed and studied (cf. [9, 10, 35] for surveys of CQA). Which notion to use may depend on the application. The notion of minimal change can be illustrated using the definition of repair given in [2]. First of all, a database instance  $D$  can be seen as a finite set of ground atoms (or database tuples) of the form  $P(\bar{c})$ , where  $P$  is a predicate in the schema and  $\bar{c}$  is a finite sequence of constants in the database domain. For example,  $Students(101, john\ bell)$  is an atom in the database. Next, it is possible to compare the original database instance  $D$  with any other database instance  $D'$  (of the same schema) through their symmetric difference  $D \Delta D' = \{A \mid A \in (D \setminus D') \cup (D' \setminus D)\}$ .

# Repair with DLV (= Datalog + ASP)

The screenshot shows the DLV IDE interface. The top menu bar includes 'File' (New, Open, Recent, Save, Print), 'Edit' (Undo, Redo, Cut, Copy, Paste, Search), 'Tools' (Preferences, Help), and a shell window. The shell window displays the command `[ASP :)$ dlv repair.dlv -filter=new_student` and its output.

```
% STUDENT(Id, Name)
student(101,bell).
student(101,ball).
student(101,logan).
student(104,stevens).
student(107,norton).

% FK: Id -> Name

% Soft constraint
icv_fk(X,Y,Z) :-
    student(X,Y), student(X,Z), Y != Z.

% Hard constraint
% :- icv_fk(_,_,_).

% REPAIR (cf. Section 3.1 [bertossi2013generic.pdf])
del_student(X,Y) v del_student(X,Z) :- icv_fk(X,Y,Z).

new_student(X,Y) :-
    student(X,Y),
    not del_student(X,Y).
```

\*shell [ASP :)\$ dlv repair.dlv -filter=new\_student  
DLV [build BEN/Dec 21 2011 gcc 4.2.1 (Apple Inc. build 5666) (dot 3)]  
{new\_student(104,stevens), new\_student(107,norton), new\_student(101,logan)}  
{new\_student(104,stevens), new\_student(107,norton), new\_student(101,ball)}  
{new\_student(104,stevens), new\_student(107,norton), new\_student(101,bell)}  
[ASP :)\$ |

• Based on [Bertossi2013]  
• If an IC is violated, let the system find all possible minimal repairs!

# Database Repair (Details)

- In this DB, the FD is violated for both
  - Id = 101 (3 times) and
  - Id = 104 (2 times),
- yielding 6 (=3\*2) possible, minimal repairs
- ASP rules and Example based on earlier reading [Bertossi2013].

The screenshot shows a DLV interface with a menu bar and toolbar. The main window displays a Datalog program for repairing student records. The code includes rules for functional dependencies (FDs) and soft constraints, and repair rules using the `del_student` predicate. The bottom part of the window shows the command-line interface where the repair program is run, resulting in 6 possible worlds.

```
% STUDENT(Id, Name)
student(101,bell).
student(101,ball).
student(101,logan).
student(104,stevens).
student(104,ftevens).
student(107,norton).

% FD (functional dependency): Id -> Name
% That is, Id is a KEY of the student table

% Soft constraint
icv_fd(X,Y,Z) :- student(X,Y), student(X,Z), Y != Z.           Record violation: distinct Y, Z for single Id X, so ...

% Hard constraint
% :- icv_fd(_,_,_).

% REPAIR (cf. Section 3.1 [bertossi2013generic.pdf])
del_student(X,Y) v del_student(X,Z) :- icv_fd(X,Y,Z).           ... mark Y or Z for deletion!

new_student(X,Y) :- student(X,Y), not del_student(X,Y).          Copy over only undeleted records

----- repair.dlv Top (10,22) (Prolog)
DLV [build BEN/Dec 21 2011 gcc 4.2.1 (Apple Inc. build 5666) (dot 3)]

{new_student(107,norton), new_student(101,logan), new_student(104,stevens)}
{new_student(107,norton), new_student(101,logan), new_student(104,ftevens)}
{new_student(107,norton), new_student(101,ball), new_student(104,stevens)}
{new_student(107,norton), new_student(101,ball), new_student(104,ftevens)}
{new_student(107,norton), new_student(101,bell), new_student(104,stevens)}
{new_student(107,norton), new_student(101,bell), new_student(104,ftevens}]

[ASP :]$ dlv repair.dlv -filter=new_student -silent | wc
6      18     452
[ASP :]$ | 6 = 3*2 possible repairs (a.k.a. "Possible Worlds") that would fix the IC violation using minimal change.

-:**- *shell* Bot (47,10) (Shell)
Wrote /Users/ludaesch/Box Sync/LIS590DCL-FA15/Datalog/ASP/repair.dlv
```

# Datalog Summary

- You might not realize it, but you now have “**the power of logic**” at your disposal:
  - **Specifying and checking** constraints (structural / schema constraints; application semantics)
  - **Datalog**: “The better First-order Logic syntax”
  - **Graph Queries, Provenance Queries** (we’ll revisit them)
- **Answer Set Programming (ASP)**
  - Datalog + AI-ish “*Generate & Test*” paradigm
  - Use it for **repairing** databases
  - cutting-edge CS research: e.g. Llunatic project