Relational Examples and Exercises

Patterns

A relation is a pattern. Naturally, we have a free choice for the syntax of the relation, given a consistent notation. Some structural alternatives include:

```
aRb
R[a,b]
(R a b)
(a R b)
```

The advantage of seeing relations as patterns is that we can then use a pattern-matching engine for abstract manipulation of patterns, that is, for computation. Pattern-matching is algebraic, permitting us to compute with unbound variables, i.e. with abstractions.

Let the `?` operator identify unbound variables, such that `?x` represents an arbitrary member of a domain (or range). Now we can express relational patterns directly in the pattern language:

reflexive	`(?x R ?x)`
symmetric	`(?x R ?y) -> (?y R ?x)`
transitive	`(?x R ?y) & (?y R ?z) -> (?x R ?z)`
antisymmetric	`(?x R ?y) & (?y R ?x) -> x=y`
trichotomy	`(?x R ?y) xor (?y R ?x) xor x=y`
irreflexive	`not (?x R ?x)`
asymmetric	`not ((?x R ?y) -> (?y R ?x))`

Binary Arithmetic Example

Relations without variables are called *facts*. The facts of binary arithmetic are:

```
(0 + 0 = 0)
(0 + 1 = 1)
(1 + 0 = 1)
(1 + 1 = 0)

(0 * 0 = 0)
(0 * 1 = 0)
(1 * 0 = 0)
(1 * 1 = 1)
```

{+,*} are functions which map pairs onto singles. This relational database has one relation, {=}. Facts are simply those relations which we assert to match our model (i.e. we consider them to be True). They define the semantics of the relation =. Specifically, the following assertions are not in the database, each is not True:

```
(0 + 0 = 1)
(0 + 1 = 0)
(1 + 0 = 0)
(1 + 1 = 1)

(0 * 0 = 1)
(0 * 1 = 1)
(1 * 0 = 1)
(1 * 1 = 0)
```

For each operator in {+,*}, the Cartesian product has eight members, four of which we distinguish as valid. The eight possible relations come from two possible values in each of three different places:

```
(A + B = C)              A,B,C = {0,1}
```

More generally,

```
(A op B R C)             op = {+,*}        R = {=}
```

A *closed world* is one in which we know that if a fact is asserted, then its negation is also asserted to be False. Binary arithmetic is a closed world. An *open world* is one in which if we know a fact, we do not necessarily know its negation. For example, if we know `(Mary isa-student)` and `(John isa-student)`, we cannot also assert `(No-one-else isa-student)`.

Pattern Generators for Binary Arithmetic

The transformational rules which define arithmetic functions can be expressed as relational patterns:

```
(?x + ?y = ?z) -> (?y + ?x = ?z)

(?x + ?y = ?z) & (?x = ?y) -> (?z = 0)

(?x + ?y = ?z) & not(?x = ?y) -> (?z = 1)
```

Pattern abstraction is available for operators as well:

```
(?x ?op ?y = ?z) -> (?y ?op ?x = ?z)
```

The above pattern generator defines a symmetry structure for both operators in our world of simple arithmetic. The operators (i.e. functions) differ by their pattern definition. Compare the constraints for + above with those for * below:

```
(?x * ?y = ?z) & (?x = 1) & (?y = 1) -> (?z = 1)

(?x * ?y = ?z) & not(?x = 1) & not(?y = 1) -> (?z = 0)
```

Pattern generators provide an alternative way to record facts into a relational database.

Option 1: explicit listing

```
(1 + 0 = 1)
(0 + 1 = 1)
```

Option 2: generator

```
(1 + 0 = 1)
(?x + ?y = ?z) -> (?y + ?x = ?z)
```

For large databases, pattern generators exchange time for space. For example, if the arithmetic example were expressed in decimal notation, we would need to assert 100 facts for each operator, out of a Cartesian product of 1000 possible facts. A symmetry rule would reduce the number of facts by half (50 here), at the cost of generating them dynamically when needed.

Using Pattern Generators for Computation

The following pattern-matching computational regime is how logic engines and theorem provers (e.g. Prolog) manage computation:

```
(0 + 0 = ?x)
```
matches
```
(0 + 0 = 0)
```
by binding `?x=0`.

Thus we have computed this Boolean sum.

Since relational structures are non-directional, we can compute inverses as well as answer more exotic queries:

```
(1 * ?x = 0)
```
matches
```
(1 * 0 = 0)
```
by binding `?x=0`.

```
(1 ?op 1 = 1)
```
matches
```
(1 * 1 = 1)
```
by binding `?op=*`

```
(1 ?op 1 = ?x)
```
matches
```
(1 * 1 = 1)
```
by binding `?op=*` and `?x=1`
```
(1 + 1 = 0)
```
by binding `?op=+` and `?x=0`

Algorithm Calling Example

Let `S = {a,b,...n}` be a set of algorithms. Define the relation `R` on `S` as

```
(a R b) iff (a CALLS b)
```

Suppose we have the following code structure:

```
(a R b)
(a R c)
(b R c)
(c R e)
(d R c)
(e R b)
```

The transitive closure of `R` identifies all algorithms which may eventually call another algorithm. Make a listing of this transitive closure. Express the closure as a pattern rule rather than an exhaustive list.

An algorithm is recursive if it calls itself, that is, if `(a R a)` is asserted. Are any of the above five algorithms `{a,b,c,d,e}` recursive?

Graph Example

Here is an airline database of costs for flying between cities

```
(San Diego to LA       $100)
(San Diego to SF       $150)
(San Diego to Portland $200)
(San Diego to Seattle  $300)
(LA to SF              $100)
(LA to Portland        $150)
(LA to Seattle         $300)
(SF to Seattle         $200)
(Portland to Seattle   $150)
```

Assume this relation is symmetric for all flights within California.

- Is it reflexive?
- Is it transitive?
- Are all combinations of flights for less than $260 transitive?
- What is the type of connectivity (1-to-1, 1-to-many, many-to-1, many-to-many) within each State?
- If the relation were antisymmetric instead of symmetric, where could you travel to?
- Which flights (could be more than one stop) are in the same cost equivalence class?
- What is the ordering of flights by cost?
- What is the transitive closure of flights?

Define a new relation `S` over the above airports to mean `in-the-same-state`. Show that S is an equivalence relation.

Text Example

Consider the relations `left-prefix` and `alphabetical-order`, for the set of alphabetic characters `{blank,a,b,...,z}`.

Build separate relational databases for each of these words:

```
dot
level
pepper
bookkeeper
```

Characterize the relational structure of each database.

Relational Database Example

Here is a relational database of students:

ID	Name	Age	Enrolled-in
29	Bob	32	512
48	John	29	514
89	Sally	28	512
59	Ignatz	34	512
30	Joan	27	510
58	Tom	24	510
45	Jane	24	514
22	Keri	28	510
97	Sam	27	512

The SELECT operation chooses row entries based on a pattern.

The PROJECT operation chooses column entries based on a pattern.

The JOIN operation creates a new entry by combining two or more relations.

Construct queries which gather the following information:

All students taking 512.
The ages of all students.
All classes with more than two students.
All students older than Joan.
Id numbers of students in each class.
Id numbers of twenty-something students in 512.