Logic and Database Queries

Moshe Y. Vardi Ian Barland Ben McMahan

August 31, 2006

Contents

1 Introduction

Databases are collections of facts, along with ways to access ("query") those given facts to discern new facts. We will be discussing the precise definition of databases, and explaining exactly what queries mean. Although one can

formalize the notion of a "collections of facts" in many ways, by far the most successful approach in computer science has been the *relational model*; this is the model which we will be spelling out below. (We won't examine other models, nor will we address other important database issues, such as how to store large databases for quick access.)

1.1 History of The Relational Model

A few of the important names in relational databases are:

- Aristotle (b. 384 BCE) and Boole (b. 1850 AD): The concept of "Properties" as a subset P of the domain $\mathcal{D}$: For example, if $\mathcal{D}$ is all people, then we can let $P_1 \subseteq \mathcal{D}$ be the set of people with one blue eye and one green eye, or let $P_2 \subseteq \mathcal{D}$ be the set of people with two blue eyes.
- Peirce (b. 1839, pronounced "purse"): Adds the concept of *relations* to propositional model, where a relation R is a set of tuples over a domain $\mathcal{D}$. For example:
  1. R could be a subset of $\mathcal{D}^1$; This is just a subset of $\mathcal{D}$. We can think of this as a property—*e.g.* all people who were born on a February 29th.
  2. R could be a subset of $\mathcal{D}^2$: This is a binary relation—*e.g.* those pairs of people who have the same birthday.
  3. R could be a subset of $\mathcal{D}^k$: This is a k-ary relation. For example, triples of people where the first two are married to each other and the third witnessed the marriage.
- Frege (b. 1848): Philosopher who wanted to formalize mathematics using logic. He recognized that any mathematical concept can be represented as a domain and a set of relations over that domain. For example, arithmetic on the natural numbers $\mathbb{N}$ might be represented as the a domain and some relations: $\langle \mathbb{N}, \leq, +, *, expt \rangle$
- Arthur Burks: Follower of Peirce. Started out as a philosopher at the University of Pennsylvania, but switched to Electrical Engineering. Worked on the ENIAC project and then moved to the University of Michigan where he founded the Computer Science department. Had Codd as a student.
- Codd (b. 1923): Student of Burks at Michigan. Once he completed his PhD there, he moved to IBM where he had the revolutionary idea that tables in a databases could be thought of as relations.

1.2 The Relational Database Model

Consider the following table, which is a database:

EmployeeName	Department	Manager
Abe	Math	Charles
Joe	CS	Jill
Zoe	Math	Charles

Here, the domain is all possible employee names, department names, and manager names. Note that there is something inelegant about throwing the values for the different fields all in the same bag, but we will come back to this later.

Now, what happens if we re-arrange the rows in the database, so that we now have this:

EmployeeName	Department	Manager
Zoe	Math	Charles
Abe	Math	Charles
Joe	CS	Jill

Intuitively, this table encodes the same facts about the world as the previous one; **the order of the rows is irrelevant**. We will consider a database to be a *set* of rows—not a list of rows—and since sets aren't inherently ordered, we get this independence from row-order for free.

Although the order of the rows does not matter, the order of the entries within a row *is* significant, of course: $\langle \text{Charles}, \text{Math}, \text{Abe} \rangle$ is not the same as $\langle \text{Abe}, \text{Math}, \text{Charles} \rangle$!

First Normal Form

What happens if you have a table where one entry might contain a *set* of values? Consider:

EmployeeName	Department	Manager	Hobbies
Joe	CS	Jill	Stamp Collecting and Rock Climbing

While this seems plausible, we are formulating the concept of a database as relations over a domain, where each table entry will correspond to exactly one element of the domain. This approach doesn't allow for having an entry containing entire lists or sets.[1] Given this constraint, how might we represent multiple hobbies? One solution is to include two rows:

EmployeeName	Department	Manager	Hobbies
Joe	CS	Jill	Stamp Collecting
Joe	CS	Jill	Rock Climbing

This solution is fine, although you might be bothered at having two rows for Joe. It's not so much that Joe occurs twice (after all, our previous example had multiple occurrences of CS and of Jill). The aesthetic rub is that the fact (relation)

[1] You might want to be clever, and say that in this example, the domain should contain employee-names, departments, and *sets*-of-hobbies. This approach certainly works, but it wouldn't be as flexible as we'll need: we could query the database for all employees whose hobbies were exactly the set $\{\text{Stamp Collecting}, \text{Rock Climbing}\}$, but not for all employees whose hobbies merely *included* Rock Climbing.

You might chafe at this restrictive model of what a database is; before proposing a more complicated model, let's work to understand this *relational database* model.

"Joe's manager is Jill" is repeated twice. The duplication isn't bothersome if this table has been synthesized from other databases (which is not uncommon; in implementation, database queries often generate intermediate tables which might have redundant information, and our theory must allow this). But if you want a single point-of-control for this information, then you might prefer to have the information stored across two different tables:

EmployeeName	Department	Manager
Joe	CS	Jill

Employee	Hobby
Joe	Stamp Collecting
Joe	Rock Climbing

This second approach is often preferred when initially defining a database, since its more aesthetically appealing (in an Occam's Razor sense) to not have duplicated—and hence possibly conflicting—facts.

Definition 1. A relational database is in **first normal form** if

- Domain values are atomic
- Relations are sets.
- Order of tuples plays no role.

In other words, in First-Normal Form, every entry in a relation is an element of the domain, and there are no sets in the domain.

1.3 The Relational Database

Now that we have explored the representation of relations in a table, we can define a relational database.

Definition 2. A **Relational Database** consists of a domain, and relations over this domain.

This definition takes into account an important distinction. Much as there is a difference between a pipe and a picture of a pipe[2], a difference between an utterance and its meaning, and a difference between a person and the name of a person, there is also a difference between a relation and the name of a relation.

For example, $<$ is the name of the less-than relation, but the set of pairs $\{\langle i, j\rangle : i, j \in N \text{ and } i < j\}$ is the actual relation[3].

Definition 3. A **relational scheme** consists of a name and an arity, k.

[2] See `http://spanport.byu.edu/Faculty/rosenbergj/Images/Magritte.Pipe.jpg`

[3] Programming languages with first-class functions often make this distinction explicit. For instance, in Scheme, `(define related? (lambda (arg1 arg2) ...))` uses `lambda` to create a function, and then uses `define` to attach a name to a value (a value which happens to be a function, in this case). Of course, creating-a-function-and-immediately-naming-it is such a common construct that the language includes syntactic sugar combining the two: `(define (related? aarg1 arg2) ...)`

Definition 4. A k-ary **relational instance** is a k-ary relation over the domain.

This allows us to make a distinction between database scheme and database instances:

Definition 5. A **database scheme** is a collection of relation schemes plus a domain.

Definition 6. A **database instance** consists of a domain and relation instances over that domain.

Think of the database scheme as the metadata, or description of the data format (how many tables there are, and the names of each table's columns). The database scheme should be very stable since changing the scheme would require changing all databases and queries. The database instance is the data itself, which can change frequently (for instance, the contents of the web).

We've included names for the relations, so that we can refer to the relations in *queries*, which actually access the data. You can think of queries as being programs.

1.4 Database Queries

Informally, a **query** is a question to the database, such as:

1. "Which department(s) is Joe in?"
2. "List the set of all professors in the Computer Science department."
3. "List the set of employees and their department whose chair is named Cher."

The answers to each of these queries will be a relation. Respectively: a unary relation (likely with only one element, at that); a unary relation with (hopefully) many elements; and a binary relation (employee-department pairs).

We formalize our notion of a query:

Definition 7. Given a database scheme S, let $DB(S)$ be all possible database instances over domain $\mathcal{D}$.

Definition 8. If S is a database scheme, then $dom(S)$ is its domain.

Definition 9. A **database query** is a question to the database that is answered by a relation of some arity k over the domain of the database. A k-ary query over database scheme S is a function $Q : DB(S) \to \mathcal{P}\left(dom(S)^k\right)$, where $\mathcal{P}\left(dom(S)^k\right)$ is the set of all possible subsets of all possible k-tuples.

While this completes the theoretical definition, there is a critical practical ingredient missing: Consider a database which relates program-names to program-texts, and the query "list all program-names such that the program-text describes a program which always terminates". Alas, the Halting Problem is not computable, so this is a query which can't actually be realized.

Remark 10 *(Desiratum).* Queries must be *computable* functions.

1.5 Query Languages

Now that we have a concept of what a query is, the next question is how do we express a query?

In the 1960s, queries were written as programs. For example, to find all of the professors who are in the Computer Science department, you would write a program that read the department column and if it matched Computer Science would then go to the corresponding professor column and output the name. This type of query writing, done in Java, C, Cobol, *etc.*, is called **imperative** or **procedural** queries because you have to tell the computer exactly what to do.

But people wanted queries to be easier to write—a higher-level language specialized for expressing queries in a way which matches the way we conceive of the questions. This led to the development of query languages like SQL ("Structured Query Language"), QUEL, and QBE. These are all **declarative** languages, where one specifies *what* characterizes an answer, without specifying *how* to compute it. The prototypical declarative language is logic—in particular, first-order logic.

We will introduce SQL in this course, but only as the culmination of a sequence: first-order logic, domain relational calculus, tuple relational calculus, and finally SQL.

We will begin by reviewing the syntax and semantics of predicate calculus (which is propositional logic plus relations), and first-order calculus (which is predicate calculus plus quantifiers).

2 Review: Predicate Calculus

In the last lecture we introduced the concept of a *relational database*. We also noted that we must make a clear distinction between data and its description—the *metadata*. The metadata of a database is called a *database scheme*. A database scheme consists of a *domain*, which is a set of relevant values (constants), and a collection of *relation schemes*, each of which has a name and arity.

The relations allow us to express and store facts about the world. For example, the ternary relation Employee$(Name, Department, Manager)$ may contain the 3-tuple $\langle$Joe, Math, Jill$\rangle$, which expresses the fact that Joe works in the Math department under Jill.

In order to access the data, we write queries, which are simply questions to the database. The answer to a query is a set of tuples. In order to study queries, we need a precise language in which to express them. For that purpose we turn to First-Order Logic.

For an more detailed introduction to these topics, see the Base Logic module of the Teachlogic project `www.teachLogic.org`. In particular, we will be review-

ing propositions[4], syntax and semantics of quantifiers[5], and semantics/interpretations[6].

2.1 Syntax of First-order Predicate Calculus

The *syntax* tells us how to build and parse well-formed formulas. It doesn't tell us how to interpret them yet: for that purpose we will define *semantics* in the next section.

Atomic Formulas

The fundamental building blocks of logical formulas are called *Atomic Formulas*. For our purposes an atomic formula is a relation from our database. Formally, we have the following

Definition 11 (Atomic Formula). An *atomic formula* is either

- $(t_1 = t_2)$, where t_1, t_2 are each a constant or a variable.
- $P(t_1, \ldots, t_k)$, where P is the name of a k-ary relation, and each t_i is either a constant or a variable for $1 \leq i \leq k, i \in \mathbb{N}$.

(A *constant* is any element of the domain.)

For example:

- Employee (Joe, Math, Jill)
  ("Jill manages Joe, who is in the math department");
- Employee (x, Math, x)
  ("x works in the math department and is their own manager").
- Employee (x, Math, y)
  ("x works in the math department and is managed by y").
  Important: observe that this formula doesn't claim x and y *have* to be different people, no more than $a + b = 4$ claims that a must be a different number from b.

Composite formulas

To build more interesting formulas we use *Boolean connectives* and *quantifiers*. Formally we have the following:

Definition 12 (Well-formed formulas). A well-formed formula ("WFF") of predicate logic is any of:

- an atomic formula, or
- $\neg\phi$, where ϕ is a WFF ("not ϕ")

[4]Propositional logic: `http://cnx.org/content/m10715/`
[5]Quantifiers: `http://cnx.org/content/m10728/`
[6]Interpretations: `http://cnx.org/content/m10726/`

- $(\phi \wedge \psi)$, where ϕ and ψ are WFFs ("ϕ and ψ")
- $(\phi \vee \psi)$, where ϕ and ψ are WFFs ("ϕ or ψ")
- $(\phi \rightarrow \psi)$, where ϕ and ψ are WFFs ("ϕ implies ψ" or "if ϕ then ϕ")
- $(\phi \leftrightarrow \psi)$, where ϕ and ψ are WFFs ("ϕ if and only if ψ")
- $\exists x.\phi$, where ϕ is a WFF ("there exists an x such that ϕ")
- $\forall x.\phi$, where ϕ is a WFF ("for all x, ϕ")

Note that the parentheses in the binary connectives—and the lack of parentheses for negation and the quantifiers—is part of the technical syntax; if you leave off parentheses or add extra ones, you no longer have an official WFF. (We won't worry much about parentheses, but it's good to be clear on what the precise definition allows.)

Example 13. The following are all WFFs. In addition to the formulas themselves, we give a brief description; this is a preview of the following section on semantics.

- $\neg\text{Employee}(\text{Zoe}, \text{Math}, \text{Jill})$ ("It's not the case that: Zoe is in the Math department and managed by Jill.")
- $(\text{Employee}(x, \text{Math}, \text{Jill}) \wedge \text{Employee}(z, \text{Math}, \text{Jill}))$ ("x and z both work in the math department and are both managed by Jill").
- $\exists y.\text{Employee}(y, \text{Math}, \text{Jill})$ ("somebody is in the math department and managed by Jill")
- $\forall x.\exists y.\text{Employee}(x, y, \text{Jill})$ ("everybody is in the same department, managed by Jill")
- $\exists y.\forall x.\text{Employee}(x, y, \text{Jill})$ ("there is a department which contains everybody, and they all are managed by Jill")

We will talk more about the precise meaning (semantics) of these formulas later.

Remark 14. The above examples have a technical problem: the quantifiers allow x (and, y) to range over both people *and* departments, not just people. So we should replace "everybody" by "everything" in the examples, which suddenly is not what we wanted. We'll address that problem eventually.

As another example, we express Fermat's Last Theorem using predicate calculus. Fermat's Last Theorem says that for any $n > 2$, there are no positive integer solutions to $a^n + b^n = c^n$. Our intended domain is positive integers, $\mathbb{N}^+$.

First we must express the exponentiation not as a function, but as a relation called *expt* with arity 3:

$$\text{expt}(x, n, z) \iff x^n = z$$

Aside: The difference between "$\leftrightarrow$" and "$\Leftrightarrow$":
The first of these two arrows is a symbol occurring within a WFF.
The second of these *never* occurs within a WFF; rather, it is a statement *about* two WFFs (asserting that they are equivalent).

Next we find a formula which expresses the notion "a, b, c are positive solutions to $a^n + b^n = c^n$."

$$\exists u.\exists v.\exists w.(\text{expt}\,(a, n, u) \wedge (\text{expt}\,(b, n, v) \wedge (\text{expt}\,(c, n, w) \wedge +\,(u, v, w))))$$

For shorthand, call this formula $\phi_{\text{Diophantus}}$. Technically, all those nested parentheses really are required, according to our official syntax for a WFF. In practice though, we'll omit the nested parentheses within the large conjunction, as well as coalescing the multiple $\exists$ quantifiers:

$$\exists u, v, w.\,(\text{expt}\,(a, n, u) \wedge \text{expt}\,(b, n, v) \wedge \text{expt}\,(c, n, w) \wedge +\,(u, v, w))$$

Just be aware that this is a lazy representation of the full WFF given above. Even when not writing out a WFF fully, be sure not to mix $\wedge$ and $\vee$ without using parentheses.

Note how we use extra "local" variables u,v,w which must correspond to a^n,b^n,c^n, in order to make the formula true. This is how we can talk about a^n in the + relation. Take a moment, to be sure you understand this.

Note that $\phi_{\text{Diophantus}}$ is a formula whose English description is stating something about the variables a, b, c, n, but *not* about u, v, w. We'll explore this important difference in a moment, when talking about the semantics of free *vs.* bound variables.

We are now ready to state Fermat's Last Theorem:

$$\neg(\exists n.>(n, 2) \wedge \exists a.\exists b.\exists c.\phi_{\text{Diophantus}})$$

"It is not the case that there is an n such that $n > 2$ and there exist a, b, c such that: a, b, c are solutions to $a^n + b^n = z^n$ (in the domain $\mathbb{N}^+$)." If you wanted to tweak the formula so that the $n > 2$ clause came *after* the next three $\exists$ quantifiers, that wouldn't change the formula's meaning.

A technical note: the Greek letter ϕ isn't occurring in our official WFF; that's only a shorthand we use for clarity[7]. The actual formula contains a total of seven "$\exists$" quantifiers.

For the mathematically inclined, here is another statement of Fermat's Last Theorem, expressed using only universal quantifiers:
$\forall n.\forall x.\forall y.\forall z.\forall u.\forall v.\forall w.((\text{expt}\,(x, n, u) \wedge (\text{expt}\,(y, n, v) \wedge (\text{expt}\,(z, n, w) \wedge >(n, 2)))) \rightarrow \neg +\,(u, v, w))$

[7] If you want to sound high-falutin', you can say that "$\phi_{\text{Diophantus}}$" is a *meta*-variable which we humans are using to represent a formula; the formula itself contains the actual logic variables such as "a" and "u."

2.2 Towards the Semantics of First-Order Predicate Calculus

Well-formed formulas are statements about the world, but the truth of such statements depends on the state of the world. For example, the statement "the nation's debt[8] is more than €22,000 per citizen" has no intrinsic truth value. Its truth value depends on the world (context) to which it refers—in this case, which nation is being discussed, as well as the particular value of the country's debt (at a certain moment in time[9]).

There is a further caveat on what formulas can mean: we'll take every statement to be either true or false—never "only partially true". Thus we won't consider statements like "The national debt is too large." This restriction known as the *Principle of the Excluded Middle*. It may not be a reasonably principle for day-to-day English, or when dealing with nuanced issues like causes of poverty, but for defining the meaning of programs, it's invaluable.

Notation

Semantics *defines* whether a formula is true or false given a description of the world. In our case, "the world" will be a database. More precisely, semantics is itself a relation "$\models$" between formulas and worlds: we denote the relation as $B \models \phi$. This reads as "B *satisfies* ϕ" or "B *models* ϕ". We can also read it from right-to-left, saying "ϕ *holds true* in B".

The semantics of a formulas like $\text{Employee}(\text{Joe}, \text{Math}, \text{Jill})$ will be easy to determine: given a database B, just look up whether the database includes the fact $\langle \text{Joe}, \text{Math}, \text{Jill} \rangle$. More precisely: we say that we're given the database $B = \left\langle \mathcal{D}, \text{Employee}^B \right\rangle$, where $\mathcal{D}$ is the domain and Employee^B is the relation instance which corresponds to the relation scheme Employee. Then the semantics of the formula $\text{Employee}(\text{Joe}, \text{Math}, \text{Jill})$ is true iff $\langle \text{Joe}, \text{Math}, \text{Jill} \rangle \in \text{Employee}^B$.

Similarly, the semantics of a formula like $>(4, 5)$ (which we accustomed to seeing in infix notation as $4 > 5$) is easy to ascertain as false. In this database view of the world, we determine this by looking up whether the actual $>$ relation instance[10] includes the pair $\langle 4, 5 \rangle$. (It doesn't!) Note that in order for $>(4, 5)$ to be a valid formula by our definition, our database's domain must include numbers.

However, we quickly run into a problem when we consider formulas of the type $>(x, 2)$ or $\text{Employee}(x, \text{Math}, x)$. These *are* a well-formed formulas according to our definition, yet even if we're given a database, we can't tell whether they hold or not, since we don't know what x is. Clearly, we will need to specify how to deal with variables.

[8] `http://www.brillig.com/debt_clock/`

[9] In math and logic, we tend to view values (sets, functions) as unchanging, and we mimic time either as a function of t, or by using temporal logic (*e.g.* see the model-checking module of `http://teachlogic.org`).

[10] Note that we'll always take the the name $>$ to correspond to the greater-than relation, and thus won't superscript the actual relation with the domain.

Variables

In order to determine the truth of formulas which involve variables, we need a way to determine the values of those variables. How can we formalize this lookup?

Definition 15 (Assignment, preliminary). An *assignment* α is a function $\alpha : \textsc{Var} \to \mathcal{D}$, where Var is the set of variables in the formula and $\mathcal{D}$ is the domain of the database.

This means that to determine the truth of a formula like $\text{Employee}(x, \text{Math}, x)$, we need to be given not only a database instance (a world), but also an assignment (a context).

Before continuing, we'll tweak this definition of an assignment. As it stands, to define the meaning of $\text{Employee}(x, \text{Math}, x)$, we say "examine each term inside the atomic formula, and if it's a variable like x then look it up in the provided assignment α; otherwise it's already an element of the domain (*i.e.* a constants like Math) and we don't need to apply α to it." We simplify this procedure applying the assignment α to both variables and raw elements of the domain, and extending assignments so that if they are given an element of the domain, they just return that element.

Definition 16 (Assignment, finalized). An *assignment* α is a function $\alpha : \textsc{Var} \cup \mathcal{D} \to \mathcal{D}$, where $\alpha(d) = d$ for all $d \in \mathcal{D}$. Var is the set of variables in the formula and $\mathcal{D}$ is the domain of the database.

Definition 17 (Single point revision). Let α be an assignment, x be a variable, and d be an element in the domain $\mathcal{D}$. Then $\alpha[x \mapsto d]$ is an assignment defined as follows:

$$\alpha[x \mapsto d](z) = \begin{cases} d & \text{if } z = x \\ \alpha(z) & \text{otherwise.} \end{cases}$$

Exercise Let α be an assignment where $\alpha(x) = \text{Joe}$, $\alpha(y) = \text{Zoe}$. What does each of the the following assignments return for the input x? For y?

- α
- $\alpha[y \mapsto \text{Joe}]$
- $\alpha[y \mapsto \text{Zoe}]$
- $\alpha[y \mapsto \text{Joe}][x \mapsto \text{Zoe}]$
- $\alpha[y \mapsto \text{Joe}][y \mapsto \text{Zoe}]$
  (Hint: this function acts just like $\alpha[y \mapsto \text{Joe}]$, except that it gives a different answer for y).
- $\alpha[y \mapsto \text{Joe}][x \mapsto \text{Joe}][y \mapsto \text{Zoe}]$

We can now give meaning to formulas with variables in them, and we extend our definition of semantics to reflect this:

Definition 18 (Semantics). Semantics is a ternary relation between a formula ϕ, a database B, and an assignment $\alpha : \text{VAR} \cup \mathcal{D} \to \mathcal{D}$, where $\mathcal{D}$ is the domain of the database. We write this as "$B, \alpha \models \phi$".

Other notations for the same idea are "$B \models_\alpha \phi$" and "$B \models \phi[\alpha]$".

2.3 Defining the Semantics of First-Order Predicate Calculus

We've said that semantics relates formulas with databases-and-assignments, but now we'll actually show how to officially calculate the semantics of a formula.

Syntax and Semantics are flip sides of the same coin. We defined the syntax of well-formed formulas: which precise sequence of characters counts as a formula. The more interesting half is semantics: how to determine the meaning of a well-formed formula. In programming terms, if you were writing a program to deal with logic (*i.e.*, with database queries!), then when writing the parser, your code would parallel our previous definition of syntax. When writing the interpreter (to give meaning to the formulas), your code would parallel the definition of semantics, below.

Parallel to our syntax definitions, first we'll define the semantics of an atomic formula, and then the semantics of compound formulas. In both cases, we'll first give some examples and figure out what we expect the semantics to be, and then we'll give the general case.[11]

Semantics of Atomic Formulas

Recall that an atomic formula is either two terms surrounding an equal-sign (like "$(4 = 6)$" or "$(4 = x)$"), or a relation applied to k terms (like "Employee $(\text{Joe}, \text{Math}, \text{Jill})$" or "Employee (x, Math, x)".

An atomic formula which doesn't contain variables is true if it in the database. In case it *does* contain variables, we must first get their values by applying the assignment. However, due to the way we defined assignments, we can apply the assignment to every element of the tuple, and magically all variables will turn into values, while the values will remain unchanged.

Remember that in the following definition, P is the *name* of a relation and P^B is the *instance* of that relation in the database B. That is, formulas can only mention the name P; the actual relation instance P^B only comes into play when discussing the semantics.

Definition 19 (Semantics of atomic formulas, simple case). Let α be an assignment, let $\langle \mathcal{D}, P \rangle$ be a database scheme[12] (where P has arity k), and let $B = (\mathcal{D}, P^B)$ be a database instance[13].

[11] This corresponds to good software design practice: start with examples of your input, then make test cases of what you expect the output to be, and only then start writing code for the general case.

[12] That is, a domain and the mere *name* "P".

[13] That is, an actual relation of arity k, over the domain $\mathcal{D}$.

Then

- $B, \alpha \models (t_1 = t_2)$ iff $\alpha(t_1) = \alpha(t_2)$.
- $B, \alpha \models P(t_1, t_2, \ldots, t_k)$ iff $\langle \alpha(t_1), \alpha(t_2), \ldots, \alpha(t_k) \rangle \in P^B$.

This definition only talks of a database with one relation, but the idea extends directly to a database with n different relations (*e.g.* Employee and Hobby and $<$). The definition only looks a bit imposing because of the subscripting:

Definition 20 (Semantics of atomic formulas, general case). Let α be an assignment, let $\langle \mathcal{D}, P_1, P_2, \ldots, P_n \rangle$ be a database scheme (where each P_i has arity k_i), and let $B = (\mathcal{D}, P_1^B, P_2^B, \ldots, P_n^B)$ be a database instance.

Then

- $B, \alpha \models (t_1 = t_2)$ iff $\alpha(t_1) = \alpha(t_2)$.
- $B, \alpha \models P_i(t_1, t_2, \ldots, t_k)$ iff $\langle \alpha(t_1), \alpha(t_2), \ldots, \alpha(t_k) \rangle \in P_i^B$, for any i in $1, \ldots, n$.

Exercise 21. Let B be a database whose domain is $\{\text{Abe}, \text{Joe}, \text{Zoe}, \text{Jill}\} \cup \{\text{Math}, \text{English}\} \cup \mathbb{N}$, let α be an assignment which maps $a \mapsto 99$ and $x \mapsto \text{Zoe}$, and let the relation $\textit{Employee}^B$ be

EmployeeName	Department	Manager
Abe	Math	Charles
Joe	CS	Jill
Zoe	Math	Charles

For each of the following formulas ϕ, determine whether or not $B, \alpha \models \phi$. In each case, what does the definition's P refer to? What does P^B refer to? (The point of this exercise isn't so much to know whether the particular formula is true or false, but to demystify the definition's notation.)

- $\phi = \text{Employee}(\text{Joe}, \text{Math}, \text{Jill})$,
- $\phi = >(4, 5)$,
- $\phi = \text{Employee}(x, \text{Math}, x)$,
- $\phi = >(n, 5)$,

So we see that if α is a mapping in which $\alpha(n) = 99$, then $B, \alpha \models\ >(n, 5)$. What about $B, \alpha[m \mapsto 314] \models\ >(n, 5)$? Here m is given a binding of 314, but it is never used in the formula, and $>(n, 5)$ still holds. We say that m is *irrelevant* to this formula, since we don't really care what α assigns to m. We'll see shortly, some formulas where m occurs in a formula, yet it might still be irrelevant!

Semantics of Composite Formulas

Now that we have defined the precise semantics of atomic formulas, we can extend that definition to composite formulas in the obvious recursive manner: For example, to see whether $(\phi \wedge \psi)$ is true for a particular database and assignment, we just check that the database/assignment makes ϕ true, and also that it independently makes ψ true.

Definition 22 (Semantics of composite formulas). Let ϕ and ψ be composite well-formed formulas, let α be an assignment, and let B be a database. Then

- $B, \alpha \models \neg\phi$ iff $B, \alpha \not\models \phi$.
- $B, \alpha \models (\phi \wedge \psi)$ iff $B, \alpha \models \phi$ and $B, \alpha \models \psi$.
- $B, \alpha \models (\phi \vee \psi)$ iff $B, \alpha \models \phi$ or $B, \alpha \models \psi$.
- $B, \alpha \models (\phi \rightarrow \psi)$ iff $B, \alpha \not\models \phi$ or $B, \alpha \models \psi$.
- $B, \alpha \models (\phi \leftrightarrow \psi)$ iff both $B, \alpha \models \phi$ and $B, \alpha \models \psi$. have the same Boolean value (it i.e. they are both true, or they are both false).
- $B, \alpha \models \exists x.\phi$ iff there exists an $a \in \mathcal{D}$ such that $B, \alpha\,[x \mapsto a] \models \phi$.
- $B, \alpha \models \forall x.\phi$ iff for all $a \in \mathcal{D}$, it is the case that $B, \alpha\,[x \mapsto a] \models \phi$.

The only interesting bits are last two cases, because they actually tweak the assignment α.

Example 23. Let B be a database whose domain is $\{\text{Abe}, \text{Joe}, \text{Zoe}, \text{Jill}\} \cup \{\text{Math}, \text{English}\} \cup \mathbb{N}$, let α be an assignment which maps $a \mapsto 99$ and $x \mapsto \text{Zoe}$, and let the relation $\textit{Employee}^B$ be

EmployeeName	Department	Manager
Abe	Math	Charles
Joe	CS	Jill
Zoe	Math	Charles

For each of the following formulas θ, determine whether or not $B, \alpha \models \theta$. In each case, identify which subformulas of θ correspond to the definition's ϕ and ψ.

- $\theta = \neg\text{Employee}\,(\text{Zoe}, \text{Math}, \text{Jill})$
  ("It's not the case that: Zoe is in the Math department and managed by Jill.")
- $\theta = (\text{Employee}\,(x, \text{Math}, \text{Jill}) \wedge \text{Employee}\,(z, \text{Math}, \text{Jill}))$
  ("x and z both work in the math department and are both managed by Jill").
- $\theta = \exists y.\text{Employee}\,(y, \text{Math}, \text{Jill})$
  ("Somebody is in the math department and managed by Jill")

- $\theta = \forall x.\exists y.\text{Employee}\,(x, y, \text{Jill})$
  ("Everybody is in the same department, managed by Jill")
- $\theta = \exists y.\forall x.\text{Employee}\,(x, y, \text{Jill})$
  ("There is a department which contains everybody, and they all are managed by Jill")

Unfortunately, as noted in Remark 14, since we have one big domain which includes both people and departments, our English equivalents should say "everything" instead of "everyone"; the universal formulas will almost always be false (which isn't what we want).

2.4 Variables: Free and Bound

We have given the syntax and semantics for first-order logic. Determining the semantics means knowing not just a domain and relations, but also an assignment for any which might variables occur. We now explore two types of variables: those which are needed to fully determine the semantics (free variables), and those which are just "local variables" to the formula (bound variables). In general: when a variable is bound to a quantifier ($\forall$ or $\exists$), then that quantifier "shadows" whatever value the assignment provided for it.

We give an example, before formalizing this notion. Consider $\phi_{\text{Diophantus}}$ from earlier:

$$\phi_{\text{Diophantus}} \equiv \exists u.\exists v.\exists w.\,(\text{expt}\,(a, n, u) \wedge \text{expt}\,(b, n, v) \wedge \text{expt}\,(c, n, w) \wedge +(u, v, w))$$

Our database instance will be standard arithmetic on positive integers: $B = \left\langle \mathbb{N}^+, \geq^B, +^B, expt^B \right\rangle$. Let α be a function which maps $a \mapsto 5$, $b \mapsto 12$, $c \mapsto 13$, and $n \mapsto 2$. Then, $B, \alpha \models \phi_{\text{Diophantus}}$, since indeed we can find values $u = 25$,$v = 144$,$w = 169$ such that $(\text{expt}\,(a, n, u) \wedge \text{expt}\,(b, n, v) \wedge \text{expt}\,(c, n, w) \wedge +(u, v, w))$. On the other hand, $B, \alpha\,[a \mapsto 99] \not\models \phi_{\text{Diophantus}}$, since there don't exist any values for u, v, w which make $(\text{expt}\,(99, 2, u) \wedge \text{expt}\,(12, 2, v) \wedge \text{expt}\,(13, 2, w) \wedge +(u, v, w))$ true.

The point is that we don't care what α does to u, v, w, since their local value will shadow whatever value α assigns to them. For example, even $B, \alpha\,[u \mapsto 999] \models \phi_{\text{Diophantus}}$. Why? By the semantics of composite formulas (the $\exists$ case), there *does* exist a value in the domain (namely, 25) such that $B, \alpha\,[u \mapsto 999]\,[u \mapsto 25] \models \phi_{\text{Diophantus}}$. So the way in which the semantics for quantifiers overrides the assignment is critical.

As a special case, consider formulas in which every variable is quantified. These formulas are either True or False in a way which depends only on the domain (the database):

- $\exists m.+\,(6, m, 4)$ ("there exists a solution to $6 + m = 4$") is true over the domain of integers, but not over merely *positive* integers.
- Similarly, $\exists z.\forall n.+\,(n, z, n)$ is true iff the domain includes a zero element for z. (In fact, this formula is used to *define* the zero element for a set.)

- This includes cases with no variables at all, like $+(2,2,4)$ (true, for the standard relation $+^{\mathbb{N}}$) and $(4=6)$ (always false).

Although you may not have thought of it this way, you have spent several years of high school dealing with formulas where the variables are *not* quantified: Algebra. In these situations, you were given a formula, and asked to come up with all assignments which make the formula true.

- For instance, "solve $2x^2+3x+4=0$" means "find all values for x such that this equation is true." (And we learn that there are no solutions making this formula true over the domain $\mathbb{R}$, but exactly two solutions over $\mathbb{C}$.)
- Moreover, when we solve $y=4x+9$, we find there are many ways to assign x and y to make the formula true, and in fact we characterize *all* possible solutions by drawing a graph—all ordered pairs which satisfy the formula. In one possible solution, $\langle -3,-3\rangle$, x and y are both assigned to the same element of the domain, and there's not the slightest problem.

Thus, we can think of equations as being queries about numbers, and algebra is a way to find all the answers to the queries. A Database class, in contrast, teaches how to solve queries over arbitrary (finite) domains.

Defining Free and Bound

The difference between variables which aren't quantified (and thus the assignment is important) and those variables which are quantified is important. Moreover, there can be multiple occurrences of a variable, some of which are quantified and some of which aren't:

$$\exists m.(<(m,n)\wedge\exists n.+(n,n,m))$$

which expresses "there is an number m which is less than n, and even". (over the domain, say, $\mathbb{N}^+$) The first occurrence of n isn't quantified, but the last occurrences *are* quantified (and they have nothing to do with the initial n).

In order to be precise, we define two new useful functions: $vars$ and $fvars$, which each take in a formula and return (resp.) its variables and its free variables. These two functions work on syntax, though we'll use them to help define semantics.

For the rest of these notes, we'll take VAR as the set of all possible variables. Thus, $\mathcal{P}(\text{VAR})$ is the power set of VAR, that is, the set of all possible subsets of VAR.

Formally: The function var takes in a formula and returns the set of variables in that formula.

$$var : \text{Formula} \to \mathcal{P}(\text{VAR})$$

The inductive definition of var is of course structured after the inductive definition of Well-Formed Formulas; we use op to refer to any of the binary connectives $\{\wedge,\vee,\to,\leftrightarrow\}$:

1. $var((t_1 = t_2)) = \{t_i : t_i \in \text{VAR}, 1 \leq i \leq 2\}$
2. $var(P\,(t_1, \ldots, t_k)) = \{t_i : 1 \leq i \leq k, t_i \in \text{VAR}\}$
3. $var(\neg\phi) = var(\phi)$
4. $var((\phi\ op\ \psi)) = var(\phi) \cup var(\psi)$
5. $var(\exists x.\phi) = var(\phi) \cup \{x\}$
6. $var(\forall x.\phi) = var(\phi) \cup \{x\}$

Similarly, $fvar$ is a function which returns the **free** variables of a formula.

$$fvar : \mathit{Formula} \to \mathcal{P}\,(\text{VAR})$$

We define $fvar$ inductively as follows:

1. $var((t_1 = t_2)) = \{t_i : t_i \in \text{VAR}, 1 \leq i \leq 2\}$
2. $fvar(P\,(t_1, \ldots, t_k)) = \{t_i : t_i \in \text{VAR}, 1 \leq i \leq k\}$
3. $fvar(\neg\phi) = fvar(\phi)$
4. $fvar((\phi\ op\ \psi)) = fvar(\phi) \cup fvar(\psi)$
5. $fvar(\exists x.\phi) = fvar(\phi) \setminus \{x\}$
6. $fvar(\forall x.\phi) = fvar(\phi) \setminus \{x\}$

Notice in the last two cases, $fvar$ is *removing* the variable being quantified.

Example 24. For example, If we look at our formula

$\phi_{\text{Diophantus}} \equiv \exists u.\exists v.\exists w.(\text{expt}\,(a, n, u) \wedge (\text{expt}\,(b, n, v) \wedge (\text{expt}\,(c, n, w) \wedge +\,(u, v, w))))$

then we have from our definitions that: $var(\phi_{\text{Diophantus}}) = \{a, b, c, n, u, v, w\}$ and $fvar(\phi_{\text{Diophantus}}) = \{a, b, c, n\}$.

Definition 25 (Sentence). A **sentence** ϕ is a well-formed formula without any free variables (*i.e.*, $fvar(\phi) = \emptyset$).

Definition 26 (Agreement). Given two assignments $\alpha_1 : \text{VAR} \to \mathcal{D}_1$, and $\alpha_2 : \text{VAR} \to \mathcal{D}_2$ and a set $X \subseteq \text{VAR}$, we say that α_1 and α_2 **agree** on X if $\alpha_1(x) = \alpha_2(x)$ for all $x \in X$.

Theorem 27 (Relevance Theorem). *Let B be a database, let ϕ be a formula, and let α_1 and α_2 be two assignments $\alpha_1 : \text{VAR} \to \mathcal{D}$ and $\alpha_2 : \text{VAR} \to \mathcal{D}$ such that α_1 and α_2 agree on $fvar(\phi)$. Then $B, \alpha_1 \models \phi$ iff $B, \alpha_2 \models \phi$.*

In words, when interpreting a formula, the only bindings you need to know are those of the free variables; you can change any of other bindings and still get the same result.

Corollary 28. *If ϕ is a sentence, A a database, and α_1, α_2 variable assignments, then $A, \alpha_1 \models \varphi$ iff $A, \alpha_2 \models \varphi$.*

(Since there are no free variables, the variable mappings trivially agree which each other.)

3 Domain Relational Calculus

We have reviewed predicate calculus, its semantics (matching relation schemes with relation instances), assignments, and free *vs.* bound variables. We now are ready to reap the benefits of all our groundwork, by seeing that *first-order formulas are queries*. This is called Domain Relational Calculus.

Definition 29. Given a database B and a formula ϕ, we define satisfy(ϕ, B) as $\{\alpha : B, \alpha \models \phi\}$.

In other words, satisfy(ϕ, B) returns all the assignments which make formula ϕ true in the database B. For example, for the formula Employee(x, Math, y) might return (depending on the particular database) two assignments, "$x \mapsto$ Abe, $y \mapsto$ Jill" and "$x \mapsto$ Zoe, $y \mapsto$ Jill". By the Relevance Theorem, we don't need to specify what the assignment does on other (irrelevant) variables.

We can generalize this example to the general case. Assume that there is some fixed order on $\textsc{Var} = \{x_1, x_2, \ldots\}$; for a formula ϕ with k free variables (that is, $k = |fvar(\phi)|$). So we have an ordering on ϕ's free variables $x_{i_1}, \ldots, x_{i_k}$. Then an assignment α can be represented as a single k-tuple over $\mathcal{D}$, which just lists the bindings of each variable, in order: $\langle \alpha(x_{i_1}), \ldots, \alpha(x_{i_k}) \rangle$. So we think of satisfy$(\phi, B)$ is a k-ary relation; by considering the database B to be an input we can view formulas as queries:

Definition 30 (Query). Let S be a database scheme. Recall that $dom(S)$ represents the domain of S and $DB(S)$ represents all database instances of S. A **query** is $Q_\phi(B) = \text{satisfy}(\phi, B)$.

So $Q_\phi : DB(S) \to \mathcal{P}\left(\mathcal{D}^k\right)$, where $k = |fvar(\phi)|$.

In short: a query takes a database and returns a relation—the set of all bindings which make formula true.

3.1 Examples of Formulas as Queries

Example 31. Consider the database scheme Employee(*EmployeeName*, *Department*, *Manager*):

1. Query: List all the departments in which Joe works for Jill.

   In first order logic: Employee$(\text{Joe}, x, \text{Jill})$.

   The free variable set is $\{x\}$, so the result is a list of all ways to assign x such that $\langle \text{Joe}, x, \text{Jill} \rangle$ is in the database's Employee relation.

2. Query: List all the employees working for Jill in Math.

   Similarly, this is Employee$(x, \text{Math}, \text{Jill})$

3. Query: List all the department-and-manager pairs for whom Joe works.

   Employee(Joe, x, y).

   This query returns a **binary** relation (a set of pairs), because the query contains two free variables, x and y. The result might be $\{\langle \text{CS}, \text{Jill} \rangle\}$.

If Joe worked in several departments, there might be more tuples in the result.

Note also that if Joe didn't work in *any* departments, the result would be the empty set, $\emptyset$, which is indeed still a list of (zero) pairs.

4. Query: List the name and department of all employees who are their own managers.

   $\text{Employee}(x, y, x)$.

   This returns a list of pairs, since $|fvars(\phi)| = |\{x, y\}| = 2$.

5. List the entire database.

   $\text{Employee}(x, y, z)$.

   This returns a list of triples (a ternary relation).

6. You might wonder whether we can write a query with *more* than three variables. Yes, although we'll need a composite formula. What does the following query ask, in English?

   $((\text{Employee}(x, d_1, z) \wedge \text{Employee}(x, d_2, z)) \wedge \neg(d_1 = d_2))$

   The result will be four-tuples of the form $\langle d_1, d_2, x, z\rangle$. (Again, if nobody meets the criteria, the result will be zero such tuples.)

   What happens if the $\neg(d_1 = d_2)$ clause is omitted?

7. We've seen queries which return relations of arity 1, 2, 3, and 4. As computer scientists, we ask: does it make sense to have queries of arity 0? Sure! We just write a query with zero free variables:

   $\text{Employee}(\text{Joe}, \text{Math}, \text{Jill})$.

   In English, "Is Joe working in Math for Jill?"

   We think of[14] the return value as either being true or false.

The take-away point: **First order formulas are declarative queries.**

3.2 More Examples of Queries

Suppose that in our database the following schemas have been defined:

- Student (*name*, *dorm*, *major*, *GPA*),
- Faculty (*name*, *dept*, *salary*, *year_hired*),

[14] Technically, though, we are still getting back a set of 0-tuples of all satisfying assignments. If false, then there are no satisfying assignments, and we get back the empty set {}. If true, we get back a set of one satisfying assignment, the 0-tuple: $\{\langle\rangle\}$. This is entirely consistent with our theory.

Unfortunately, often languages like SQL don't distinguish the empty set from the set containing the empty tuple. They are reduced to hacking the solution by saying queries of no free variables return a boolean, not a relation. Sigh.

- Chair ($dept$, $name$),
- Teachers ($name$, $course$), and
- Enrolls ($name$, $course$).

We will look at several examples that illustrate how queries can be written.

1. List the name, dorm, and major of students with a GPA of exactly 3.0:

$$\text{Student}\,(n, c, m, 3.0)\,.$$

2. List name, dorm, and major of students with a GPA at least 3.0.

   Attempt 1:

$$(\text{Student}\,(n, c, m, g) \wedge \geq (g, 3.0)).$$

   This is *nearly* correct, but has one significant problem[15]: As written, the query has four free variables, which means it will return 4-tuples as answers, even though we only want 3-tuples.

   The solution to this is to quantify out g:

$$\exists g.(\text{Student}\,(n, c, m, g) \wedge \geq (g, 3.0)).$$

3. List all the students and the dorm they belong to.

$$\exists m, g.\text{Student}\,(n, c, m, g)\,.$$

   In this example, as in example 2, we use $\exists$ to "quantify out" columns so that they are not free variables and thus not returned.

4. List names and departments of faculty who were hired before 1980 and make a salary less than $50000.

$$\exists s.\exists y.(\text{Faculty}\,(n, d, s, y) \wedge (< (s, 50000) \wedge < (y, 1980))).$$

5. List names of faculty members and their chairs. The fundamental difference here is that the information is not stored in a single table; we must combine 2 tables. Suppose we write

[15] A minor technicality is that our query uses the relation named $\geq$, yet that isn't one of the schemas given to us.

Really, this was an oversight in the database scheme. For this class we will assume that the relation schemas such as $<$, $\leq$, $\geq$, and $>$ (for numbers) are always built-in to the database, and that they always have their standard relation instances ("they always mean what you think they mean"). In addition, $=$ is also always available (not just for numbers, but for all values), by our definition of atomic formulas.

We will often use these relations with infix notation notation, and will also use $(a \neq b)$ as an abbreviation for $\neg\,(a = b)$

$$(\text{Faculty}\,(n, d, s, y) \wedge \text{Chair}\,(d, n_1))$$

Notice that the d is the same in both tables. This query returns a 5-ary tuple, but we only wanted 2 items. Thus, we need to quantify out the rest:

$$\exists d.\exists s.\exists y.(\text{Faculty}\,(n, d, s, y) \wedge \text{Chair}\,(d, n_1)).$$

6. List names of faculty members whose salary is higher than that of their chair.

   Here, all the information we need is contained in two tables, but we have to access the *Faculty* table twice to get the chair's info. The only free variable will be n, for the faculty member's name.

$$\exists d, s, y, n_1, d_1, s_1, y_1.(\text{Faculty}\,(n, d, s, y) \wedge (\text{Chair}\,(d, n_1) \wedge (\text{Faculty}\,(n_1, d_1, s_1, y_1) \wedge > (s, s_1)))),$$

   Remember, $\exists d, s, y, n_1, d_1, s_1, y_1 \ldots$. is a short-hand for $\exists d.\exists s.\exists y.\exists n_1.\exists d_1.\exists s_1.\exists y_1 \ldots$..

7. List the names of faculty who have the highest salary in their department.

$$\exists d, s, y.(\text{Faculty}\,(n, d, s, y) \wedge \forall n_1, s_1, y_1.((\text{Faculty}\,(n_1, d, s_1, y_1) \rightarrow \geq (s, s_1)))).$$

   Because of logic equivalence rules[16] we can rewrite this as:

$$\exists d, s, y.(\text{Faculty}\,(n, d, s, y) \wedge (\neg\exists n_1, s_1, y_1.(\text{Faculty}\,(n_1, d, s_1, y_1) \wedge < (s, s_1)))).$$

Abstracting: Three types of queries

We observe that the above queries have certain aspects that can be classified as follows:

- *Filter*, which can be thought of as "erasing rows" in a table, but otherwise returning the entire (non-erased) rows. Example 1 is a perfect example, although filtering plays a substep in the other queries.
- *Select*, which can be thought of as "erasing columns" in a table. Example 3 is a pure example of this, although again nearly every example above also includes some selecting.
- *Join*, which combines different relations to form new ones. Examples include queries 5, 6, 7. Note that in query 7, *Faculty* is joined with itself.

[16] Propositional equivalances: `http://cnx.org/content/m10717/`; first order equivalences: `http://cnx.org/content/m10729/`

4 Tuple Relational Calculus

The style which we have been using to write queries is known as *Domain Relational Calculus*, abbreviated "DRC," since every variable ranges over the values in the domain. There is another style called *Tuple Relational Calculus*, abbreviated "TRC," which is used to write queries. We highlight the similarities and the differences between the two styles.

There are two significant shortcomings with the way we have treated relational databases so far. First, we assume that there is only a single domain $\mathcal{D}$, comprised of every possible interesting value, *e.g.*, people, departments, integers, and floating-point numbers. So when we have the relation scheme Student $(name, dorm, major, GPA)$, there is nothing which stops us from inserting the tuple $\langle 1, 2, 3, Joe \rangle$ into the relation. It would makes sense to require that names, for example, all conform to the type (say) `CHAR[64]`. Second, currently we have viewed relations as sets, so the order of tuples is irrelevant. Yet within a tuple, the order of entries (columns) are inherently relevant. For example, in the *Student* relation, the user has to remember that the first column represents the name, the second column the dorm, *etc.* It would be nice to have a formalism to access the columns by name, rather than by order.[17]

To incorporate these improvements, we will tweak our definition of the relational model.

1. We assume a set $Types$ of types. With each type $t \in Types$ there is an associated domain $\mathcal{D}_t$ of values of this type. For example, the type `CHAR[64]` is the domain of character strings with length less than 64. We can now take the over-arching domain $\mathcal{D}$ to be $\bigcup_{t \in Types} \mathcal{D}_t$.

2. So far a relation scheme has been just a name and an arity. We now assume that there is a set *Attr* of *attributes* (column names). A relation scheme S consists of a name and a set of typed attributes (at typed attribute is a pair, consisting of an attribute and a type). For example, a relation scheme for the *Student* relation can be:
   Student (*name* `char[64]`, *dorm* `char[64]`, *major* `char[64]`, *GPA* `float`).

3. So far a k-ary tuple instance has been an element of $\mathcal{D}^k$. Suppose now we have a relation scheme S, consisting of a name, say R, and a set $\{(a_1, t_1), \ldots, (a_k, t_k)\}$ (where a_i is the ith column's name, and t_i its type). A tuple instance with respect to S is a mapping $\tau : \{a_1, \ldots, a_k\} \to \mathcal{D}$ such that $\tau(a_i) \in \mathcal{D}_{t_i}$ for $i = 1, \ldots, k$. Note that in this definition there is no order for the attributes of a relation. A relation instance over S is a set of tuple instances over S. Thus, neither rows nor columns are ordered.

In other words, a tuple is a set of assignments of values to attributes, each of which type-checks appropriately. Some examples of tuples for the relation:

[17] In programming languages, that's the difference between a structure (which has named fields), and an array (which has numbered fields).

Student (*name* `char[64]`, *dorm* `char[64]`, *major* `char[64]`, *gpa* `float`)

Name	Dorm	Major	GPA
Johnny	Will Rice	CS	3.9
Janie	Lovett	Biology	3.8
Jehosaphat	Baker	PolySci	3.7

A Relation Instance is then a set of tuples, where each field can be accessed by name (as discussed).
A Database Scheme consists of types, domains, and relational schemas.
A Database Instance consists of types, domains, and a set of relational instances (one for each relation scheme).

4.1 Syntax and Semantics of Tuple Relational Calculus

The biggest difference between DRC and TRC well-formed formulas is that in the TRC, variables represent tuples, and a tuple assignment α returns not elements of the domain, but tuples of the domain. Moreover, we will need to extract attributes from a tuple; we denote this with an infix dot: $t.name$ extracts the attribute *name* from a tuple variable t. (Hopefully, in the semantics, t will be assigned to a tuple which actually has the desired attribute!)

One quick example of a TRC formula, before launching into the official definition:
List the students who have a GPA of 3.0:

$$(\text{Student}\,(t) \wedge t.gpa = 3.0)$$

This formula is satisfied by an assignment α and database instance A such that $\alpha(t)$ is one of the all the tuples in the relation instance Student^A, and $\alpha(t)$ has the *gpa* attribute 3.0.

Definition 32 (Well-Formed Formula (TRC)). A Well-Formed Formula of the Tuple Relational Calculus, and its semantics, is any of the following cases:

- (Atomic Formula: tuple assignment)
  $P(t)$, where P is a relation and t is a tuple variable.
  $A, \alpha \models P(t)$ iff $\alpha(t) \in P^A$.

- (Atomic Formula: comparison to constants)
  $(t.a\ comp\ c)$, where t is a tuple variable, a is an attribute, c is a constant in the domain, and *comp* is one of $=, <, \leq, \geq, >$.
  $A, \alpha \models (t.a\ comp\ c)$ iff $\alpha(t)$ has an attribute[18] named a, and $\alpha(t).a\ comp\ c$.

[18] Observe that $t.gpa \neq 3$ isn't the same as $\neg t.gpa = 3$, since tuples without a gpa attribute at all satisfy the second but not the first. These tuples will also satisfy $\neg(t.gpa \neq 3 \vee t.gpa = 3)$ and (perhaps unexpectedly) $\neg t.gpa = t.gpa$. Thus, $\forall t.t.gpa = t.gpa$ isn't a tautology.

We *want* to say "attributes can only be applied to tuples which which actually contain that attribute; otherwise the formula isn't syntactically correct." But this is easier said than defined:

- (Atomic Formula: comparison between tuple attributes)
  $(t.a\ comp\ s.b)$, where s, t are tuple variables, a, b are attributes, and *comp* is one of $=, <, \leq, \geq, >$.
  $A, \alpha \models (t.a\ comp\ s.b)$ iff $\alpha(t)$,$\alpha(s)$ have attributes named (resp.) a,b, and $\alpha(t).a\ comp\ \alpha(s).b$.

- (Composite Formula: boolean connectives)
  $(\phi \wedge \psi)$, where ϕ,ψ are well-formed TRC formulas.
  $A, \alpha \models (\phi \wedge \psi)$ iff $A, \alpha \models \phi$ and $A, \alpha \models \psi$.

  The other boolean connectives ($\neg$, $\vee$, $\rightarrow$, $\leftrightarrow$) are defined correspondingly.

- (Composite Formula: quantifiers)
  $\exists t.\phi$ and $\forall t.\phi$, where ϕ is a well-formed TRC formula.
  $A, \alpha \models \exists t.\phi$ iff for some tuple instance $\overline{a}$ we have $A, \alpha\,[t \mapsto \overline{a}] \models \phi$.

  The semantics of $\forall t.\phi$ is defined correspondingly.

4.2 Examples

Database Schema for use in examples:

STUDENT (*name* `char[64]`, *dorm* `char[64]`, *major* `char[64]`, *GPA* `float`)

FACULTY (*name* `char[64]`, *dept* `char[64]`, *salary* `long int`, *year_hired* `int`)

CHAIR (*dept* `char[64]`, *name* `char[64]`)

TEACHES (*name* `char[64]`, *course* `char[64]`)

ENROLLS (*name* `char[64]`, *course* `char[64]`)

4.3 A Selection List

Earlier, when we asked for a TRC formula for students with a 3.0 GPA, we wrote $(\text{STUDENT}\,(t) \wedge (t.gpa = 3.0))$. But, this returns a full tuple for each t matching GPA. If we wanted only the name and dorm of each student, we need some way to select just those columns. We do this with a selection list:
$\langle t.name, t.dorm \rangle\ (\text{STUDENT}\,(t) \wedge (t.gpa = 3.0))$
(This is sometimes called *projection*, since it takes the four-dimensional tuple and projects it into a two-dimensional space.)

– This new requirement isn't context-free, making a formal definition non-trivial.

– We need to specify some algorithm for determining when an attribute is guaranteed to be in a tuple. For example, should

$$(\forall t.(\neg\text{ENROLLED}\,(t) \rightarrow \text{STUDENT}\,(t)) \wedge (\neg\text{ENROLED}\,(t) \wedge (t.gpa = \ldots)))$$

be allowed? How to encode assumptions about the database schema, that (say) some relations must include others?

Nonetheless, we will officially wave our hands: A formula can only reference attributes on tuples which can be proven to contain that attribute.

We introduce a slight distinction between a formula and a query, with the difference being that queries select attributes from satisfying assignments:

Definition 33 (TRC Query). A *query* in the Tuple Relational Calculus has the form $\langle t_i.a_j, \ldots, t_k.a_l \rangle \, \phi$, where ϕ is a well formed TRC formula, and $t_i, \ldots, t_k$ are free variables in ϕ.

The result of the query Q for the database A is: $Q(A) = \{\langle \alpha(t_i).a_j, \ldots, \alpha(t_k).a_l \rangle : A, \alpha \models \phi\}$

We give some familiar examples in our new language:

1. List the name and dorm of CS students with a GPA of at least 3.0:

   $$\langle t.name, t.dorm \rangle \, (\text{STUDENT}\,(t) \wedge ((t.major = CS) \wedge (t.gpa \geq 3.0)))$$

   For comparison, in DRC this was: $\exists z.(\text{STUDENT}\,(x, y, CS, z) \wedge \geq (z, 3.0))$

2. List the names of faculty members with a salary of at most 50,000 who were hired before 1980:

   $$\langle t.name \rangle \, (\text{FACULTY}\,(t) \wedge ((t.salary \leq 50000) \wedge (t.year\_hired \leq 1980)))$$

3. List the names of students who take courses from their department chair:

   $$\begin{array}{lll} \langle t_1.name \rangle & (& (\text{STUDENT}\,(t_1) \wedge \text{CHAIR}\,(t_2) \wedge \text{TEACHES}\,(t_3) \wedge \text{ENROLLS}\,(t_4)) \\ & \wedge & ((t_1.major = t_2.dept) \wedge (t_2.name = t_3.name) \wedge (t_3.course = t_4.course) \wedge (t_1.name = t \end{array}$$

   Remark 34. The conjunctive clauses can be written in any order, without changing the meaning; you can choose any order which makes sense to you. However, when we segue to SQL, we'll see that it is convenient to group all the tuple selections together, followed by the comparisons which involve particular attributes.

4. List the names of faculty whose salary is higher than their chair's salary:

   $$\begin{array}{lll} \langle t_1.name \rangle & (& (\text{FACULTY}\,(t_1) \wedge \text{CHAIR}\,(t_2) \wedge \text{FACULTY}\,(t_3)) \\ & \wedge & ((t_1.dept = t_2.dept) \wedge (t_2.name = t_3.name) \wedge (t_1.salary > t_3.salary)) \quad) \end{array}$$

5. List the names of faculty members whose salary is highest in their department:

   $$\langle t_1.name \rangle \, (\text{FACULTY}\,(t_1) \wedge \forall t_2.((\text{FACULTY}\,(t_2) \wedge (t_1.dept = t_2.dept)) \rightarrow (t1.salary \geq t_2.salary)))$$

   Remark 35. In general, queries written for TRC use the quantifiers much less often than queries written for DRC. However, this example shows that quantifiers can still be useful in TRC.

5 From DRC and TRC to SQL

In our previous TRC queries, we saw that they each had three components:

- a select list, such as $\langle t.name, t.dorm, t.major \rangle$, followed by
- a range, such as $(\text{STUDENT}\,(t_1) \wedge \cdots \wedge \text{TEACHES}\,(t_k))$, followed by
- filtering: criteria such as $((t.gpa \geq 3.0) \wedge \cdots)$.

We'll see in a moment that SQL makes all three components explicit with the keywords `SELECT`, `FROM`, `WHERE` (resp.).

5.1 Initializing an SQL database

Before we write SQL queries, we'll see how to create a database and insert tuples into it. You will want to try this out on an SQL implementation, such as PostgreSQL.

These three commands are used to create the database and to insert the data:

```
CREATEDB my_very_own_database
CREATE TABLE STUDENT(name CHAR[64], dorm CHAR[64], ...)
INSERT INTO STUDENT(name, dorm, major, gpa) VALUES (Joe, WillRice,
CS, 3.2)
```

When we create the table using `CREATE TABLE`, we construct the columns in a particular order. PostgreSQL saves this as the default order of the columns. If we don't specify explicitly an order when we add data to the table, PostgreSQL assumes that we are using the default order. For example, either of the following also insert the same record into the database:

```
INSERT INTO STUDENT VALUES (Joe, WillRice, CS, 3.2)
INSERT INTO STUDENT(gpa, dorm, name, major) VALUES (3.2, WillRice,
Joe, CS).
```

5.2 Examples of SQL queries

To write a query, we use this format:

```
SELECT t.name, t.dorm, t.major
FROM STUDENT t
WHERE t.gpa > 3.0
```

- `SELECT` is our select list.
- `FROM` is the tables which we are searching
- `WHERE` is the filtering criterion.

1. List name of faculty member who makes more than $50k and were hired before 1980.

   ```
   SELECT t.name
   FROM FACULTY t
   WHERE t.salary > 50000 AND t.year_hired < 1980
   ```

   Note that the `WHERE` statement uses Boolean connectives, expressed in English. Also, if there is no ambiguity, we can remove the tuple variable:

   ```
   SELECT name
   FROM FACULTY t
   WHERE salary > 50000 AND year_hired < 1980
   ```

2. List names of students who take a course from their chair

   ```
   SELECT t1.name
   FROM STUDENT t1, CHAIR t2, ENROLLS t3, TEACHES t4
   WHERE t1.major = t2.dept AND t2.name = t4.name AND t4.course =
   t3.course AND t1.name = t3.name
   ```

3. List names of faculty whose salary is higher than their chair's:

   ```
   SELECT t1.name
   FROM FACULTY t1, CHAIR t2, FACULTY t3
   WHERE t1.dept = t2.dept AND t2.name = t3.name AND t1.salary >
   t3.salary
   ```

If `SELECT *` means select everything; if `SELECT` (with no parameters) means select nothing.

5.3 TRC *vs.* SQL

Can all TRC queries be translated into SQL? That is, is SQL as expressive as TRC?

Consider this well-formed TRC query:

$$\langle t.name \rangle \left(\neg \text{STUDENT}\,(t) \wedge (t.name = Jones) \right)$$

This is an odd query: even though the query only mentions students, we can a thousand non-students to the database, and the query's results might suddenly change! In general, this is considered very poor manners. The problem arises from the use of the negation.

This TRC query does *not* have an SQL counterpart, as SQL doesn't allow negations in the `WHERE` clause. So in general SQL is not as expressive as TRC. Characterizing its exact expressiveness is a topic we don't cover here. However, we do define a few concepts and state some theorems of interest:

Definition 36 (Compatibility). A query q is *compatible* with a database scheme S if q mentions only relation names, attribute names, and values from S.

Compatability is a natural notion; you don't want queries which mention relations and attributes which aren't actually in your database.

Definition 37 (Domain Independence). A query q is *domain independent* if, for all database schemes S_1 and S_2 which are compatible with q and differ only on the domains, then for any database instances B_1,B_2 of S_1,S_2 (resp.) such that B_1 and B_2 have precisely the same relation-instances for all relations mentioned in q, then $q(B_1) = q(B_2)$

Domain independence formalizes the notion that if we start with a database B_1, and modify it to become B_2—but only using relations and tuples which the query q doesn't mention—then this modification doesn't change what q returns.

Theorem 38. *SQL can express all domain-independent TRC queries. Furthermore, all SQL queries are domain-independent.*

So even though SQL strictly less expressive than TRC, this restriction can actually be seen as an advantage, and TRC is arguably *too* expressive:

Theorem 39. *Checking domain-independence of TRC queries is undecidable.*