More first-order syntax

Professor Cian Dorr

9th October 2022

New York University

Relational first-order languages (review)

A relational signature is a set of strings (not containing special characters, not variables), together with an arity function assigning an arity to each of them, e.g.:

$$[\texttt{Loves} \mapsto 2, \texttt{Wise} \mapsto 1]$$

Call it . Here are some members of $\mathcal{L}()$, the set of first order formulae over :

```
Wise(x)∧(Loves(x,y)∧¬Loves(y,x'''))
```

```
 x  y (Wise(x) Loves(x,y))
```

```
(Wise(x)  x Wise(x))
```

Unofficially:

$$Wise(x) \simeq Loves(x, y) \simeq \neg Loves(y, x''')$$

$$-x - y(Wise(x)\ \ Loves(x, y))$$

$$Wise(x)\ \ *xWise(x)$$

A perspective on unofficial notation

Sample recursive definitions for relational formulae

Definition

FV is the function $\mathcal{L}(\Sigma) \to \mathcal{P}(\mathrm{Var})$ such that

1. For any atomic formula $P = R(v_1, \ldots, v_n)$, $FV(P) = \{v_1, \ldots, v_n\}$.
2. For any formula P, $FV(\neg P) = FV(P)$.
3. For any formulae P, Q, $FV(P \simeq Q) = FV(P \Leftarrow Q) = FV(P\ Q) = FV(P) \Uparrow FV(Q)$
4. For any formula P and variable v, $FV(-vP) = FV(*vP) = FV(P) \setminus \{v\}$.

We say that P is *closed*, (a *sentence* of $\mathcal{L}(\Sigma)$) when $FV(P) = \varnothing$; otherwise *open*.

Intuitively: if we have a particular way of *understanding* $\mathcal{L}(\Sigma)$, the sentences are going to be the ones that we can use to *assert that something is the case*. *Open* formulae by contrast are more analogous to *predicates* in English; e.g. $\mathrm{loves}(x, x) \approx$ `loves itself'.

Capture-free substitution

Example

For any variables u, v, $[u/v] : \mathcal{L}(\Sigma) \to \mathcal{L}(\Sigma)$ is the largest partial function such that:

(1) $R(v_1, \ldots, v_n)[u/v] = R(u_1, \ldots, u_n)$ where $u_i = v_i$ if $v_i \neq x$ and $u_i = y$ if $v_i = x$

(2) $(\neg P)[u/v] = \neg(P[u/v])$

(3) $(P \# Q)[u/v] = P[u/v] \# Q[u/v]$ for $\# = \simeq, \Leftarrow,$

(4) $$(-wP)[u/v] = \begin{cases} -wP & \text{if } v = w \\ -w(P[u/v]) & \text{if } v \vdash w \text{ and } (u \vdash w \text{ or } v \notin FV[P]) \\ \text{undefined} & \text{if } v \vdash w \text{ and } u = w \text{ and } v \in FV[P] \end{cases}$$

(5) $$(*wP)[u/v] = \begin{cases} *wP & \text{if } v = w \\ *w(P[u/v]) & \text{if } v \vdash w \text{ and } u \vdash w \text{ or } v \notin FV[P] \\ \text{undefined} & \text{if } v \vdash w \text{ and } u = w \text{ and } v \in FV[P] \end{cases}$$

Capture-free substitution

The function defined on the previous slide is called *capture-free substitution of u for v*.

Its definition depends on a minor extension of the recursion theorem which allows for partial functions. We can recover this from our recursion theorem by defining it as a total function on $\mathcal{L}() \cup \{\texttt{()undefined()}\}$, then restricting to $\mathcal{L}()$.

- Why do we care about this rather messy function? The answer is that we are going to want to use the following rules to capture reasoning about quantifiers:

  $$- E : P[u/v] \text{ follows from } -vP$$
  $$* I : *vP \text{ follows from } P[u/v]$$

  But $*x\mathrm{Loves}(x,x)$ should not follow from $-y*x\mathrm{Loves}(x,y)$!
  Also, $\mathrm{Wise}(x) \simeq *y\mathrm{Loves}(x,x)$ should not follow from $-y(\mathrm{Wise}\, y \simeq *y\mathrm{Loves}(y,y)$.

Adding function symbols

First-order signatures

We are going to be interested in first-order languages that are suited for talking about *numbers* and *strings*.

One could formalize theories in arithmetic using a relational signature. For example, we could talk about addition using a three-place predicate Sum where $\mathrm{Sum}(x, y, z)$ means `x and y add up to z'. We could express the commutativity of addition using a sentence

$$-x_1 - x_2 - y(\mathrm{Sum}(x_1, x_2, y) \ \mathrm{Sum}(x_2, x_1, y))$$

and its associativity with

$$-x_1 - x_2 - x_3 - z(*y(\mathrm{Sum}(x_1, x_2, y) \simeq \mathrm{Sum}(y, x_3, z)) \ *y(\mathrm{Sum}(x_2, x_3, y) \simeq \mathrm{Sum}(x_1, y, z)))$$

But this gets really awkward.

First-order signatures

For such purposes, it's a lot more convenient to use a more general kind of first-order language that contains *function symbols* as well as predicates. If we talk about addition with a binary function symbol $+$, we can express commutativity and associativity as one would expect, as:

$$-x-y(x+y=y+x)$$
$$-x-y-z(x+(y+z)=(x+y)+z)$$

First-order signatures

Definition

A *first-order signature* is a triple $\Sigma = \langle R_\Sigma, F_\Sigma, a_\Sigma \rangle$, where R_Σ and F_Σ are two *non-overlapping* sets of strings that don't contain any special characters and are not in Var, and $a_\Sigma : (R \Uparrow F) \to \mathbb{N}$.

When $\Sigma = \langle R_\Sigma, F_\Sigma, a_\Sigma \rangle$ is a first-order signature, i is called a *predicate* of Σ if $i \in R_\Sigma$ and $a_\Sigma i > 0$; a *function symbol* of Σ if $i \in F_\Sigma$ and $a_\Sigma i > 0$; a *sentence constant* of Σ if $i \in R_\Sigma$ and $a_\Sigma i = 0$; and an *individual constant* of Σ if $i \in F_\Sigma$ and $a_\Sigma i = 0$.

Example

The *language of arithmetic* is the signature Arith with $R_{\text{Arith}} = \{\leq\}$, $F_{\text{Arith}} = \{0, \mathtt{suc}, +, \times\}$, $a(\leq) = 2$, $a(0) = 0$, $a(\mathtt{suc}) = 1$, $a(+) = 2$, $a(\times) = 2$.

First order languages with function symbols

Introducing individual constants and functional symbols makes the definition of *atomic formula* more complicated: we first define a different set of strings $\mathrm{Terms}(\Sigma)$, and use it to define $\mathrm{Atoms}(\Sigma)$.

Definition

When Σ is a first-order signature, $\mathrm{Terms}(\Sigma)$ is the smallest set X of strings such that:

1. $v \in X$ for all $v \in \mathrm{Var}$.
2. $c \in X$ whenever c is an individual constant of Σ.
3. $f(t_1, \ldots, t_n) \in X$ whenever f is a function symbol of Σ with $a_\Sigma f = n$, and $t_1, \ldots, t_n \in X$.

Unique readability for terms

Notice that we use parentheses and commas to demarcate a function symbol and its arguments. Again, this is to avoid ambiguity and secure

Unique Readability for Terms

The set Terms(Σ), considered as an algebraic structure, has the Injective Property.

That is: the functions $([\langle t_1, \ldots, t_n\rangle \mapsto f(t_1, \ldots, t_n)])_{f\in F_\Sigma}$ are all injective, and their ranges do not overlap one another or Var or the set of individual constants of Σ.

This can be proved in exactly the same way as unique readability for relational formulae, by showing that every term is balanced and every initial substring of a term is either parenthesis-free or left-heavy.

Atomic formulae

We need new definitions of `atomic formula' and `formula' that allows terms that aren't variables to occur as arguments of predicates:

Definition

Where Σ is a first-order signature, $\mathrm{Atoms}(\Sigma)$ is the set of all strings of the form

$$i \oplus \texttt{(} \oplus t_1 \oplus \texttt{,} \oplus \cdots \oplus t_n \oplus \texttt{)}$$

where each $t_i \in \mathrm{Terms}()$ and either $i \in R_\Sigma$ and $n = a_\Sigma i$, or $i = \texttt{=}$ and $n = 2$.

The definition of `formula' for first-order signatures is exactly the same as for relational signatures, just plugging in the new definition of atomic formula.

Unique readability

We can prove the following using the same techniques as before

Unique Readability for Atomic Formulae

For any signature Σ, any predicates $F, G \in R_\Sigma \cup \{=\}$ such that $a_\Sigma(F) = n$ or $F = =$ and $n = 2$ and $a_\Sigma(G) = m$, or $G = =$ and $m = 2$, and any $t_1, \ldots, t_n, t'_1, \ldots t'_m \in \mathrm{Terms}(\Sigma)$, if $F(t_1, \cdots, t_n) = G(t'_1, \cdots, t'_m)$ then $F = G$ and $n = m$ and $t_k = t_k$ for each k.

Unique Readability Theorem for First-Order Formulae

When Σ is a first-order signature, $\mathcal{L}(\Sigma)$ (considered as an algebraic structure) has the Injective Property. That is, on the set of formulae of $\mathcal{L}(\Sigma)$, the functions $[s \mapsto \neg s]$, $[\langle s, t\rangle \mapsto (s \wedge t)]$, $[\langle s, t\rangle \mapsto (s \vee t)]$, $[\langle s, t\rangle \mapsto (s \to t)]$, $([s \mapsto \forall v\, s])_{v \in \mathrm{Var}}$ and $([s \mapsto \exists v\, s])_{v \in \mathrm{Var}}$ are all injective, and their ranges do not overlap each other or $\mathrm{Atoms}(\Sigma)$.

Recursive definitions for terms

Since Terms(Σ) has the Inductive and Injective properties, we have a version of the Recursion Theorem for terms, and can use it to give recursive definitions such as the following.

Definition

Vars is the function $\mathrm{Terms}(\Sigma) \to \mathcal{P}\,\mathrm{Var}$ such that:

1. For any variable v, $\mathit{Vars}(v) = \{v\}$.
2. For any individual constant c, $\mathit{Vars}(c) = \varnothing$.
3. For any n-ary function symbol f and terms $t_1, \ldots, t_n$, $\mathit{Vars}(f(t_1, \ldots, t_n)) = \mathit{Vars}(t_1) \cup \cdots \cup \mathit{Vars}(t_n)$.

Recursive definitions for terms

Definition

For any variable v and $t \in \mathrm{Terms}(\Sigma)$, $[t/v]$ is the unique function $\mathrm{Terms}(\Sigma) \to \mathrm{Terms}(\Sigma)$ such that

1. $v[t/v] = t$
2. $u[t/v] = u$ when u is a variable other than v.
3. $c[t/v] = c$ when c is an individual constant.
4. $f(t_1, \ldots, t_n)[t/v] = f(t_1[t/v], \ldots, t_n[t/v])$.

(Again we follow the tradition of writing this sort of function in postfix position.)

Recursive definitions for formulae, redux

A common pattern for defining a function from the set of formulae of a first-order signature Σ to some domain is to first recursively define a helper function from terms, then use it to define a function for atomic formulae, and finally recursively extend that to all formulae. For example:

Definition

FV is the function $\mathcal{L}() \to \mathcal{P}\,\mathrm{Var}$ such that

1. For an atomic formula $P = R(t_1, \ldots, t_n)$, $FV(P) = \mathrm{Vars}\, t_1 \cup \cdots \cup \mathrm{Vars}\, t_n$.
2. For any formula P, $FV(\neg P) = FV(P)$.
3. For any formulae P, Q, $FV(P \simeq Q) = FV(P \Leftarrow Q) = FV(P\ Q) = FV(P) \Uparrow FV(Q)$
4. For any formula P and variable v, $FV(-vP) = FV(*vP) = FV(P) \setminus \{v\}$.

Capture-free substitution, redux

Definition

For any term t and variable v, $[t/v] : \mathcal{L}(\Sigma) \to \mathcal{L}(\Sigma)$ is the partial function such that:

$$R(v_1, \ldots, v_n)[t/v] = R(t_1, \ldots, t_n) \text{ where } t_i = v_i \text{ if } v_i \neq v \text{ and } t_i = t \text{ if } v_i = v$$

$$(\neg P)[t/v] = \neg(P[t/v])$$

$$(P \# Q)[t/v] = P[t/v] \# Q[t/v] \text{ for } \# = \simeq, \Leftarrow,$$

$$(-uP)[t/v] = \begin{cases} -uP & \text{if } v = u \\ -u(P[t/v]) & \text{if } v \vdash u \text{ and } (u \notin FV(t) \text{ or } v \notin FV[P]) \\ \text{undefined} & \text{if } v \vdash u \text{ and } u \in FV(t) \text{ and } v \in FV[P] \end{cases}$$

$$(*uP)[t/v] = \begin{cases} *uP & \text{if } v = u \\ *u(P[t/v]) & \text{if } v \vdash u \text{ and } (u \notin FV(t) \text{ or } v \notin FV[P]) \\ \text{undefined} & \text{if } v \vdash u \text{ and } u \in FV(t) \text{ and } v \in FV[P] \end{cases}$$

First-order languages with definite descriptions

Note: in a couple of weeks, we will introduce one more convenient expansion of first-order syntax, namely the addition of a *definite description operator*.

This is not crucial (any more than it's crucial to have function symbols rahter than predicates), but will help simplify some definitions relevant to the ``definability'' and ``representability'' of functions.