Objects

Propositional logic is interesting on its own, but logic gets a lot more useful when we can use it to talk about properties of objects in real or imaginary worlds. Starting in this section, we'll show how to include objects to get a new, more general reasoning system called *predicate logic*.

An example of the kind of objects we might want to reason about are integers: we define an object for each integer, along with functions like $+$ and $\times$ that combine them. Each object can have multiple names: e.g., the name x and the name 7 could refer to the same object. We then provide inference rules to work with our objects, like the fact that $\times$ distributes over $+$.

Finally, we provide a way to make true-false statements (propositions) about our objects: we define *predicates* that act on objects, like equality and inequality $x = y$ or $z \geq 0$. Once we have these propositions, we can treat them just like any other logical propositions: we can combine them using $\vee, \wedge, \neg, \rightarrow$, and apply inference rules like modus ponens or double-negation elimination. For example, if we had facts

$$x = y \qquad y \geq 0 \qquad (x = y) \wedge (y \geq 0) \rightarrow (x \geq 0)$$

we could use $\wedge$-introduction and modus ponens to conclude $x \geq 0$.

Because of the distinction between objects and truth values, there are two types of expressions in predicate logic:

- terms, which evaluate to objects, and
- propositions, which evaluate to true or false.

The innermost quantities in any expression of predicate logic are terms. We can combine terms to make compound terms like $4 \times (x + 3)$. Eventually we apply predicates to make propositions like $4 \times (x + 3) \geq 0$. We can think of applying predicates as moving to the outer layer of our expression; in this outer layer we combine truth values using connectives like $\vee$ or $\wedge$. This two-layer structure is in contrast to propositional logic, where every subexpression is a proposition, and there is no division into layers.

Objects and types

Each object can belong to one or more named types like int or char. Based on their types, we can combine objects using a few simple constructions.

First, we can make tuples of objects, and we can index into these tuples. For constructing tuples we'll use commas like $(a, b, \mathrm{Spot}, 7)$, and for indexing we'll use functions like $\mathrm{first}(3, 7) = 3$ and $\mathrm{second}(\mathrm{Mary}, \mathrm{Spot}) = \mathrm{Spot}$. The type of a tuple is a *product type*: e.g., the type of $(7, \text{‘q’})$ is $\mathrm{int} \times \mathrm{char}$.

Second, we can apply functions: e.g., the function $+$ is an object of type $\mathrm{int} \times \mathrm{int} \to \mathrm{int}$, so we can apply it to a pair of ints to produce another int. We can also define new functions using λ-abstraction: e.g., $\lambda x, y, z.\, (x + y) \times z$ defines a function of type $\mathrm{int} \times \mathrm{int} \times \mathrm{int} \to \mathrm{int}$.

> *Note that we use the same symbol, $\to$, for function types and for implication. And, we use the same symbol, $\times$, for product types and for multiplication. The distinctions should be clear from context.*

Finally, we can define and work with disjoint union types like $\mathrm{int} \mid \mathrm{char}$. We build values of a union type by typecasting: $[\mathrm{int} \mid \mathrm{char}]\, 7$ and $[\mathrm{int} \mid \mathrm{char}]\, \text{‘q’}$ both have the same type. And we use values of a union type in case statements: if x is of type $\phi \mid \psi$, and f and g are functions of type $\phi \to \chi$ and $\psi \to \chi$, then $(f \mid g)\, x$ is of type χ. This expression applies the function f if x happens to be of type ϕ, and g if x happens to be of type ψ.

Every type can be used as a predicate: e.g., $\mathrm{int}(7)$ is true, while $\mathrm{char}(7)$ is false.

We define distinguished types any and none; all objects have type any, and no objects have type none. So, $\mathrm{any}()$ is a synonym for T (it tests whether the empty tuple has type any), and $\mathrm{none}()$ is a synonym for F.

Example: natural numbers

The above operations for working with tuples, unions, and functions can be thought of as built into predicate logic. For any particular use of predicate logic, we'll add problem-specific objects, types, and inference rules. In this section we'll give an example: a logical system for working with natural numbers.

First we define a type, $\mathbb{N}$, and an element of that type, 0.

Next we define a predicate, $=$, and rules for working with equality:

- Equality takes two arguments of type any. We'll use the typical infix notation $a = b$

instead of $=(a, b)$, and we'll write $a \neq b$ as a shorthand for $\neg(a = b)$.

- Equality is reflexive: if ϕ is any term, then $\phi = \phi$.
- Equality is symmetric: if ϕ and ψ are terms, then from $\phi = \psi$ we can conclude $\psi = \phi$.
- Equality is transitive: for terms ϕ, ψ, χ, if $\phi = \psi$ and $\psi = \chi$, then $\phi = \chi$.

Our last rule for working with equality will be the most complex; it lets us substitute equal objects for one another. We'll need some notation: suppose ϕ is a term, and suppose ψ is any expression that contains ϕ as a subexpression. Write $\psi[\phi \to \chi]$ for the expression that results by replacing ϕ with χ inside of ψ.

- Substitution: if ψ is an expression containing the term ϕ, and if χ is another term, then from ψ and $\phi = \chi$, we can conclude $\psi[\phi \to \chi]$.

For example, from $\mathbb{N}(x)$ and $x = y$, we can conclude $\mathbb{N}(y)$.

Next we define the successor function S, which has type $\mathbb{N} \to \mathbb{N}$, along with rules for working with successors:

- For terms ϕ and ψ of type $\mathbb{N}$, from $\phi = \psi$ we can conclude $S(\phi) = S(\psi)$, and from $S(\phi) = S(\psi)$ we can conclude $\phi = \psi$.
- For any term ϕ of type $\mathbb{N}$, we can conclude $S(\phi) \neq 0$.

Finally, we define the rule of mathematical induction:

- For any predicate p, if we can prove $p(0)$ and $p(x) \to p(S(x))$ (where x is a fresh, otherwise unused variable), then we can conclude $p(\phi)$ for any term ϕ of type $\mathbb{N}$.

Example: addition is commutative

The above definitions let us prove lots of interesting facts about natural numbers. A good example is to show that addition is commutative.

Define a function $+$ of type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ using the following inference rules:

- For any term ϕ of type $\mathbb{N}$, $\phi + 0 = \phi$.
- For any terms ϕ, ψ of type $\mathbb{N}$, $\phi + S(\psi) = S(\phi + \psi)$.

To make expressions easier to read, we'll use infix notation for $+$. We'll use ordinary numbers as shorthand, like $2 = S(S(0))$.

We can see that $+$ impements addition: e.g.,

$$\begin{aligned} 3+2 &= S(3+1) \\ &= S(S(3+0)) \\ &= S(S(3)) \\ &= 5 \end{aligned}$$

Here we've applied substitution on $2 = S(1)$, used the second rule for $+$ to move the S outside in $3 + S(1) = S(3+1)$, and the first rule for $+$ together with substitution to get rid of $+0$ in the subexpression $3+0$.

Given the above definitions, our first step in showing that addition is commutative is to show $a + 0 = 0 + a$. We can do this by induction. We write our induction hypothesis as a predicate: $p(a) \leftrightarrow (a + 0 = 0 + a)$. (Here $\leftrightarrow$ is short for implication in both directions.)

We first have to prove $p(0)$: $0 + 0 = 0 + 0$ since equality is reflexive.

We then have to prove that $p(a) \rightarrow p(S(a))$:

$$\begin{aligned} a + 0 &= 0 + a \\ S(a+0) &= S(0+a) \\ S(a) &= S(0+a) \\ S(a) + 0 &= S(0+a) \\ S(a) + 0 &= 0 + S(a) \end{aligned}$$

We'll leave the remainder of the proof as an exercise:

> Use induction on the predicate $q_a(b) \leftrightarrow (a + b = b + a)$ to show $a + b = b + a$.

For more details about data types and functions, like λ-abstraction and union types, see the notes from 10-606, starting at 1/19.