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 $+$.

We also 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 $z \geq 0$. (We can still have bare propositions like q or happy, as in propositional logic; we think of these as predicates that take no arguments.) Once we have constructed propositions by applying predicates to objects, 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 $x \geq 0$.

> *Just like before, we'll use either single characters or strings of characters as names for objects and predicates. The only change is that now we disallow parentheses inside names, since we need parentheses for function application and predicate application. We'll use context to disambiguate whether a name refers to an object or a predicate.*

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. We can *declare* the type of any term to be clear:

$$7 : \text{int} \qquad \text{`q'} : \text{char} \qquad (3.2 + 4.1) : \mathbb{R}$$

We'll be informal and skip declarations whenever they are clear from context. We will assume though that we have a specific type in mind for every term.

> *A complete logical system typically requires type declarations on at least some terms. Most systems also give rules for **type inference** so that they can tell some terms' types without a declaration.*

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 $(a, b, \text{Spot}, 7)$, and for indexing we'll use functions like $\text{first}(3, 7)$ and $\text{second}(\text{Mary}, \text{Spot})$. The type of a tuple is a *product type*: e.g., the type of $(7, \text{`q'})$ is $\text{int} \times \text{char}$.

Second, we can apply functions: e.g., the function $+$ is an object of type $\text{int} \times \text{int} \to \text{int}$, 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 $\text{int} \times \text{int} \times \text{int} \to \text{int}$. Recall that λ-abstraction *binds* object names: e.g., the name x is bound in $\lambda x.\, x + y$, while the name y is free (not bound). We call an object name a *variable* if we bind it or intend to bind it.

> *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. We'll also use $\to$ for one more purpose below (substitution). The distinctions should be clear from context.*

Finally, we can define and work with disjoint union types like $\text{int} \mid \text{char}$. We build values of a union type by typecasting: $[\text{int} \mid \text{char}]\, 7$ and $[\text{int} \mid \text{char}]\, \text{`q'}$ both have the same type. And we use values of a union type via 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 is equivalent to $f(x)$ if x happens to be of type ϕ, and to $g(x)$ if x happens to be of type ψ.

*We make the convention that unions are **tagged**, i.e., we remember which type they are actually holding. And, we require case statements to be exhaustive, i.e., there has to be exactly one case for each possible type in the union. With these two conventions, it is never ambiguous how to interpret a case statement.*

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.

Substitution

To describe some of our inference rules below, we'll need a syntactic process called *substitution of terms*.

Suppose ϕ is a term; for example, ϕ could be the expression x. Suppose ψ is an expression that contains ϕ as a subexpression; for example, ϕ could be $(x+3)\times y$. Let χ be another term — say $\chi = 2z$.

Given the above, we write $\psi[\phi \to \chi]$ for the expression that results by replacing ϕ with χ inside ψ. In our example, $((x+3)\times y)[x \to 2z]$ yields $(2z+3)\times y$. If ϕ appears multiple times inside ψ, we can replace a single instance of ϕ or several; we just need to say which we mean. (If not specified, we'll assume we mean all applicable instances.)

Substitution is only applicable to free names. That is, none of the names that appear free in ϕ or χ can be bound at the substitution point (the point where ϕ appears inside of ψ). For example, if ψ is $\lambda x.\, \lambda y.\, (x+y+z)$, we can substitute $z \to a$ in ψ, but we can't substitute $(y+z) \to a$ (since y is bound at the substitution point in ψ), and we can't substitute $z \to (x+2)$ (since x would become bound at the substitution point in ψ). A free name in ϕ that becomes bound when we substitute into ψ is said to be *captured*, so we can summarize the last rule as "no capturing."

As an example use of substitution, we can define a parameterized family of propositions. Let ψ be a propositional formula that contains a free variable x. Let ϕ be a term. Then we write $\psi(\phi)$ for $\psi[x \to \phi]$. Here we replace all free instances of x; and as usual we disallow capturing variables in ϕ. E.g., if ψ is $2\times x+3 = 5+x$, then $\psi(2)$ is $2\times 2+3 = 5+2$ (which is true), while $\psi(9)$ is $2\times 9+3 = 5+9$ (which is false).

Below we'll be able to express this same idea more simply using quantifiers.

Simplifying terms

As hinted in our definitions above, we have inference rules for simplifying terms. For product types $S \times T$, we may use:

- Product reduction: we can replace the expression $\mathrm{first}(\phi, \psi)$ by ϕ. Similarly, we can replace the expression $\mathrm{second}(\phi, \psi)$ by ψ. The same is true for third, fourth, ... in longer tuples.

For union types $(S \mid T)$, we may use:

- Union reduction: If χ has type S, we can replace the expression $(\phi \mid \psi)(\chi)$ by $\phi(\chi)$. If χ has type T, then we can replace by $\psi(\chi)$.

For function types $S \to T$, we may use:

- Renaming: if ϕ is a term and y is any object name that does not appear in ϕ, we can replace the expression $\lambda x.\, \phi$ by $\lambda y.\, \phi[x \to y]$. That is, we can rename the bound variable.
- Function reduction: if χ has type S and ϕ has type T, then we can replace the expression $(\lambda x.\, \psi)(\chi)$ by $\psi[x \to \chi]$. Here we replace all free occurrences of x in ψ, and any free variables in χ may not become bound at any of the substitution points.

Note that we might need to rename in order to enable function reduction. E.g., taking an example from the previous section, we can't function-reduce $(\lambda x.\, \lambda y.\, (x + y + z))(y + z)$ because y is bound in the function expression. But if we first rename to $(\lambda x.\, \lambda a.\, (x + a + z))(y + z)$, we can function-reduce to $\lambda a.\, (y + z + a + z)$.

> *Renaming is sometimes called α-conversion, and function reduction is sometimes called β-reduction.*

Example: pairs

As a simple example of how to work with the above inference rules, we can re-implement pairs (tuples of length 2) using λ-expressions. This isn't necessarily useful, since we're considering tuples to be a built-in construct already, but it's good practice.

Define pair to be the term $\lambda x, y.\, \lambda f.\, f(x, y)$. Define first to be the term $\lambda x, y.\, x$, and define second to be the term $\lambda x, y.\, y$. And suppose we have a base type T (the type of x and y above), with some members John, Sue, Spot.

Then we can interpret $\text{pair}(\text{Spot}, \text{John})$ to be the pair with members Spot and John. This interpretation works because

$$\begin{aligned}&(\text{pair}(\text{Spot}, \text{John}))(\text{first})\\ &= (\lambda f.\, f(\text{Spot}, \text{John}))(\text{first})\\ &= \text{first}(\text{Spot}, \text{John})\\ &= \text{Spot}\end{aligned}$$

The first equality is function reduction on pair. The second is function reduction on the λf expression. And the last is function reduction on first.

Exercise: evaluate $(\text{pair}(\text{Spot}, \text{John}))(\text{second})$.

Example: natural numbers

The above operations for working with tuples, unions, and functions can be thought of as built into predicate logic. For any significant use of predicate logic, we'll add problem-specific objects, types, and inference rules. In this section we'll give a moderate-sized 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.

- Substitution of equals: if ψ is an expression containing the term ϕ, and if χ is another term, then from ψ and $\phi = \chi$, we can conclude $\psi[\phi \to \chi]$. If ϕ appears multiple times, we can substitute any number of occurrences. As usual, none of the free object names in ϕ or χ may be bound at any of the substitution points.

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:

- Equality of 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 $\phi = \psi$.
- Zero is the smallest natural number: for any term ϕ of type $\mathbb{N}$, we can conclude $S(\phi) \neq 0$.

Finally, we define the rule of mathematical induction. Induction works on a parameterized family of propositions $p(x)$, called the *induction hypothesis*.

- Induction: suppose we can prove $p(0)$ and $p(x) \to p(S(x))$ (where x is a fresh variable). Then we can conclude $p(\phi)$ for any term ϕ of type $\mathbb{N}$.

Here $p(0)$ is called the *base case*, and $p(x) \to p(S(x))$ is called the *inductive step*. A *fresh* variable is one that doesn't appear anywhere else in our assumptions or conclusions. (It's OK if the variable appears bound somewhere that doesn't conflict with its use here, although it's typically clearest to choose a name that doesn't appear at all.)

Induction is an extremely general and useful principle. We defined it here for the natural numbers, but we will see it again later in other contexts.

> *This is not the strongest possible form of induction. We won't go into detail, but the stronger forms imply that all natural numbers are reachable from 0 using the successor function S. Our weaker form only says that, if $p(x)$ is true for all reachable x, it is true for all x.*

To demonstrate induction, we can prove that all numbers are either even or odd. Define even and odd as follows:

- Zero and one are even and odd, $E(0)$ and $O(S0)$.
- $E(\phi) \to E(SS\phi)$ for each term ϕ of type $\mathbb{Z}$.
- $O(\phi) \to O(SS\phi)$ for each term ϕ of type $\mathbb{Z}$.

Then we can prove all numbers are even or odd by induction. Pick the induction hypothesis $p(x)$ to be

$$(E(x) \vee O(x)) \wedge (E(Sx) \vee O(Sx))$$

For the base case: the conclusion

$$(E(0) \vee O(0)) \wedge (E(S0) \vee O(S0))$$

follows from the definitions $E(0)$ and $O(S0)$.

For the inductive step: from the induction hypothesis $p(x)$, we have $E(x) \vee O(x)$, which implies $E(SSx) \vee O(SSx)$ using proof by cases. The induction hypothesis also gives us $E(Sx) \vee O(Sx)$. Putting these together with $\wedge$-introduction, we get

$$(E(Sx) \vee O(Sx)) \wedge (E(SSx) \vee O(SSx))$$

which is $p(Sx)$.

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:

- Zero is the additive identity: for any term ϕ of type $\mathbb{N}$, we can conclude $\phi + 0 = \phi$.
- Relationship of addition to successor: for any terms ϕ, ψ of type $\mathbb{N}$, we can conclude $\phi + S(\psi) = S(\phi + \psi)$.

To make expressions easier to read, we'll use infix notation for $+$. And, we'll use ordinary numbers as shorthand, like $2 = S(S(0))$. We'll also often combine pairs of related steps: instead of separately concluding $a + 0 = a$ and using substitution of equals to replace $a + 0$ by a in a larger expression, we will do both in a single line.

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 the shorthand $2 = S(1)$, used the second rule for $+$ (relationship of $+$ and S) to move the S outward in $3 + S(1) = S(3 + 1)$, and used the first rule for $+$ (additive identity) 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 $p(a)$, defined as $a + 0 = 0 + a$.

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

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

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

The first line above is $p(a)$. The second line uses equality of successors. The third and fourth lines use that 0 is the additive identity. The last line uses the relationship between addition and the successor function; this line is our desired conclusion, $p(S(a))$.

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

> Use induction on the predicate $q_a(b)$, defined as $a + b = b + a$, to show that addition is commutative.

Quantifiers

So far, when we've needed to talk about properties of many objects, we've used inference rules like "if ϕ is any term, we can conclude $P(\phi)$". This can be inconvenient since we need a lot of inference rules.

Instead, we can use *quantifiers* to talk about multiple objects at once. We'll give inference rules for two quantifiers, $\exists$ and $\forall$.

We interpret the formula $\forall x : T. P(x)$ as "for all x of type T, the proposition $P(x)$ holds." The introduction rule for $\forall$ is

- Universal introduction: suppose ϕ is a propositional formula that contains a free variable x of type T. And suppose that x is fresh. Then, if we can prove ϕ, we can conclude $\forall x : T. \phi$.

In this new conclusion, $\forall x$ binds the previously-free occurrences of x in ϕ. We will sometimes omit the type T if it is clear from context, and write just $\forall x. \phi$. Recall that a fresh variable is one that doesn't appear free anywhere in our previous assumptions and conclusions.

The intuition for the $\forall$-introduction rule is that, if we can prove that something is true about an arbitrary object x of type T, it must be true for all objects of the same type.

The elimination rule for $\forall$ is:

- Universal elimination: from $\forall x : T. \phi$, for any term ψ of type T, we can conclude $\phi[x \to \psi]$. As usual, we replace every free occurrence of x in ϕ, and any free variables in ψ may not be captured.

In order to apply $\forall$-elimination without capturing free variables, we sometimes need

- Renaming: if y is a fresh variable of type T, then from $\forall x : T. \psi$ we can conclude $\forall y : T. \psi[x \to y]$.

The interpretation of these rules is straightforward: if we know that a property holds for all objects of type T, we can conclude it for any specific object of type T.

We interpret the formula $\exists x : T. P(x)$ as "there exists at least one object x of type T for which $P(x)$ holds." The introduction rule for $\exists$ is

- Existential introduction: suppose ϕ is a propositional formula that contains a free object name y of type T. Let x be an object name that doesn't occur free in ϕ. Then, if we can prove ϕ, we can conclude $\exists x : T. \phi[y \to x]$.

In this new conclusion, $\exists x$ binds the occurrences of x in $\phi[y \to x]$. We will sometimes omit the type T if it is clear from context, and write just $\exists x. \phi$.

The interpretation for $\exists$-introduction is that, if we can prove the property ϕ for some specific object, then there must exist at least one object that satisfies ϕ.

Note that the only difference between $\forall$-introduction and $\exists$-introduction is that, in the latter, the object name y is not required to be fresh: we're free to prove the property ϕ about some object that is already mentioned in our previous assumptions and conclusions. Because our previous assumptions and conclusions might tell us something about y, we only know that the property ϕ holds for y, and not for arbitrary objects of the same type.

The elimination rule for $\exists$ is

- Existential elimination: let y be a fresh object name of type T. Then from $\exists x : T. \phi$, we can conclude $\phi[x \to y]$.

The interpretation of this rule is that we can give a name to the object that we know exists. (The same object may have other names already, but we can always give a new one.)

Quantifiers and inference rules

Now that we have quantification, we can re-state some of our previous ideas more succinctly. For example, instead of defining a parameterized family of propositions using substitution, we can define it and give it a name within our logical system. Before, we would have started from a propositional formula like

$$(\text{brother}(\text{father}(x)) = y) \vee (\text{brother}(\text{mother}(x)) = y)$$

and used substitution on x and y to define a parameterized family that we might call $\text{uncle}(x, y)$. Now, we can directly write in first-order logic:

$$\forall x, y.\, \text{uncle}(x, y) \leftrightarrow [(\text{brother}(\text{father}(x)) = y) \vee (\text{brother}(\text{mother}(x)) = y)]$$

(Here $p \leftrightarrow q$ is short for $(p \to q) \wedge (q \to p)$.)

Inference rules can become simpler as well: for example, induction becomes

- Induction: let p be a predicate. Suppose we can prove $p(0)$ and $\forall x : \mathbb{N}.\, p(x) \to p(S(x))$. Then we can conclude $\forall x : \mathbb{N}.\, p(x)$.

In this new formulation, we have one version of the inference rule for each predicate name, instead of one version for each natural number and each member of a parameterized family of propositions.

Quantifiers and functions

Quantification is a powerful tool. For this reason, it is common to place restrictions on its use. One of the most important questions is whether we allow quantification over function types.

If we *disallow* quantification over functions, that means that we can only quantify over base types and types built from base types using products and unions. The resulting system is called *first-order* predicate logic.

If we *allow* quantification over functions, we get *second-order* logic. In this case it's common also to allow quantification over predicates. (The two are equivalently powerful:

a function whose output type is $\{0, 1\}$ is equivalent to a predicate.)

But, second-order logics can be tricky: without some care, it's possible to create a logic that is *inconsistent*, i.e., where some formula has to be both true and false. One common way to avoid paradoxes is to remove some of the inference rules that we've defined above. For example, if we remove double-negation elimination (and its related rules like the law of the excluded middle), we get the basis for a consistent, constructive, second-order logic.

To avoid these problems, we will stick to first-order logic. First-order logic is strictly less expressive, but it is sufficient for most purposes (and will be sufficient for ours).

> *One place where first-order logic's limitations are inconvenient is in handling induction. In first-order logic we can't say directly that induction works for all predicates; instead we have to resort to language like "if p is a predicate name, then the following rule holds." And, in first-order logic we can't say something like "the valid natural numbers are precisely those that we can reach from 0 by applying the successor function repeatedly." Instead we have to use the weaker form of induction given above. But these limitations are not too harmful, in the sense that induction in first-order logic is still highly expressive.*

Unification

The inference rules that we've given so far are sufficient to define first-order predicate logic. But there are other useful rules that we can derive from the ones we've given. One of the most common and useful derived rules is *resolution with unification*.

Resolution with unification works similarly to the plain resolution rule that we introduced for propositional logic. Just as before, we start by reducing all of our assumptions to a conjunction of clauses, and we apply resolution to pairs of clauses to conclude new clauses.

In first-order logic, though, a clause has a slightly more general form: it looks like

$$\forall x, y, z.\, (p(x, y) \vee \neg q(\mathrm{Fred}, f(z)) \vee r())$$

That is, each clause is a disjunction of literals, and each literal is a possibly-negated proposition, just like before. But now, each proposition is a predicate applied to zero or more terms. The terms can contain object names like Fred. They can contain function applications like $f(z)$. And they can contain universally-quantified variables like x.

> *There's no need for existentially-quantified variables; we eliminate them through a process called*

> ***Skolemization**, which we won't cover here.*

In order to use resolution, we need to find a matching literal: one that appears positively in one clause and negatively in another. In propositional logic, matching literals were easy to find: two literals matched if and only if they were textually identical. In first-order logic, on the other hand, two literals can match even if they look different. They match when:

- Their predicate is the same, and
- For each argument position, the term in that position from the first literal *unifies* with the corresponding term from the second literal.

Two terms ϕ and ψ are said to *unify* if they could potentially refer to the same object. For example, the terms $\text{father}(\text{Sue})$ and $\text{father}(x)$ can unify, since they can refer to the same object if we substitute $x \rightarrow \text{Sue}$.

More generally, there's an algorithm that can determine, for any two terms, what is the least-restrictive set of substitutions that we have to make so that they are guaranteed to refer to the same object. (We won't go into this algorithm here; it's enough for us to know it exists.) Before we perform resolution, we apply the computed substitutions, so that the positive and negative literals become textually identical.

Once we generalize to resolution with unification, we get the same completeness property as we had for propositional logic: if ϕ is any first-order formula that follows from our assumptions, then there is a resolution proof of ϕ by contradiction. That is, if we add $\neg\phi$ to our assumptions, there is a resolution proof of F.

> *If our logic includes $=$, we need inference rule of substitution of equals in addition to resolution and unification.*

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