Logic and computation

First-order predicate logic, as we've defined it so far, is very expressive. To measure how expressive, one approach is to look at the kinds of computations it can encode.

For one example, it's easy to encode a finite automaton in first-order logic: we assume types char and string, with functions $\mathrm{first}(s : \mathrm{string})$ and $\mathrm{rest}(s : \mathrm{string})$ that return the first character and all but the first character, respectively. We define a type state and a predicate $\mathrm{reachable}(s : \mathrm{state})$. We assert $\mathrm{reachable}(s_0, t)$ for an initial state s_0 and a string t; the idea is that we want to test whether t is accepted by the automaton. For each possible transition in our automaton, if the transition goes from $s : \mathrm{state}$ to $s' : \mathrm{state}$ and consumes a character $q : \mathrm{char}$, we assert

$$(\mathrm{reachable}(s, t) \wedge \mathrm{first}(t) = q) \rightarrow \mathrm{reachable}(s', \mathrm{rest}(t))$$

The idea is that the second argument of the $\mathrm{reachable}$ predicate tells us what part of the string remains to be processed.

We finally test acceptance by asserting $\mathrm{final}(s)$ for final states s, and checking whether

$$\exists s.\, \mathrm{reachable}(s, \epsilon) \wedge \mathrm{final}(s)$$

where ϵ is the empty string. The valid strings t are the ones for which we can prove acceptance: that is, the ones that provably satisfy

$$\mathrm{reachable}(s_0, t) \rightarrow \exists s.\, \mathrm{reachable}(s, \epsilon) \wedge \mathrm{final}(s)$$

Since we can encode a finite automaton, we know that first-order logic can accept at least regular languages. But in fact we can go much further: we won't prove it here, but a generalization of the above construction can simulate a Turing machine. So, first-order logic can accept r.e. languages — that is, first-order logic can accomplish any computation that we believe to be possible in any formalism.

First-order logic is not, however, the most convenient programming language. In the remainder of this set of notes, we'll try to remedy one of its shortcomings: we'll add a way to define new, expressive types called *inductive*

types.

> *The idea that logic can encode computation is an old one. It turns out that there is a very deep connection between these two ideas: effectively any feature of a logical language can be translated into a feature of a programming language, and vice versa. This connection is called the* ***Curry-Howard correspondence****.*

Inductive types

Realistic programs don't just work with integers: they define and work with complex types like trees and heaps. These are called *inductive* types, since we reason about them using a version of induction (defined below). Instances of inductive types are constructed recursively; e.g., big trees are defined by putting together littler trees.

To reason about inductive types, we need appropriate definitions and inference rules. We'll define an inductive type by providing a list of *constructors*, i.e., ways to build a value of that type. For example, we can define a binary tree with two constructors:

$$\textbf{type}\ \mathrm{tree} = \mathrm{leaf}(v : \mathrm{int}) \mid \mathrm{node}(l : \mathrm{tree}, r : \mathrm{tree})$$

This definition says that a tree can either be a leaf containing a single int or a node containing a pair of trees. Each of these trees can then itself be a single int or a pair of trees, and so on. For example, a tree could be $\mathrm{node}(\mathrm{leaf}(7), \mathrm{node}(\mathrm{leaf}(6), \mathrm{leaf}(5)))$, which looks like this:

The constructors leaf and node become new functions. Their output types are $\mathrm{leaf}(\mathrm{int})$ and $\mathrm{node}(\mathrm{tree}, \mathrm{tree})$. These new types are defined to be subtypes of tree; in fact, tree behaves like a union type, with one case per constructor. The constructor names are arbitrary, but each constructor must be distinct from the others (either by name or by argument types).

> *In a fully-detailed logical system, we'd want to give convenient and expressive scoping rules for type names, and possibly allow mutually-recursive type declarations. To keep*

things simple, we'll instead just require all type declarations to happen in the global scope, and we'll say that each type declaration may reference itself and any previously declared types. Duplicate names shadow previous uses, and can be shadowed by subsequent uses.

Every inductive type T must have at least one *base case*: a constructor that doesn't take any objects of type T as input. The constructors that aren't base cases are called *inductive*.

*Our intent is that each instance of an inductive type should be **finite**; that is, the recursion must eventually bottom out by choosing a base case constructor. We'll say more below about this property.*

To use a tree, we have to access its components, with potentially different behavior depending on how the tree was constructed. To access components, we'll generalize the λ syntax for function definition: we allow functions like

$$\lambda\,\mathrm{leaf}(v : \mathrm{int}).\, v$$

The function header $\lambda\,\mathrm{leaf}(v : \mathrm{int})$ means that we take an argument of type $\mathrm{leaf}(\mathrm{int})$ and bind v to the int stored inside. Similarly, we could have a header $\lambda\,\mathrm{node}(l : \mathrm{tree}, r : \mathrm{tree})$ which extracts l and r from an argument of type $\mathrm{node}(\mathrm{tree}, \mathrm{tree})$. This kind of argument syntax is often called a *destructuring bind*, since it pulls arguments out from inside a specified structure.

The process for handling different constructors is very much like the process we've already seen for accessing a union type: we need to treat each constructor as a separate case. To do this, we'll generalize the case statement: we define one function to handle each constructor, and switch between them depending on the actual type of the argument.

For example, let's write a function sum of type $\mathrm{tree} \to \mathrm{int}$ to calculate the sum of all the leaves in a tree. At a leaf we just want to return the value stored there. At an internal node we want to recursively apply sum to the left and right children, and add the results.

$$\begin{aligned}
&\mathrm{sum} = \lambda(t : \mathrm{tree}).\,(\\
&\quad \lambda\,\mathrm{leaf}(v : \mathrm{int}).\, v \mid\\
&\quad \lambda\,\mathrm{node}(l : \mathrm{tree}, r : \mathrm{tree}).\, \mathrm{sum}(l) + \mathrm{sum}(r)\\
&)\,t
\end{aligned}$$

The requirements for this new kind of case statement are the same as what we

had above for cases on ordinary union types: the functions we combine with $|$ all have to have the same return type, and there must be exactly one case for each constructor. In our example, the return type for each case is int. The two cases cover the two ways to construct a tree: as $\text{leaf}(\text{int})$ or as $\text{node}(\text{tree}, \text{tree})$.

To go with these new kinds of syntax, we have corresponding reduction rules. We'll state them informally, since they are essentially the same as the rules for function reduction and union reduction.

The rule for destructuring bind is:

- Destructuring function reduction: we apply $\lambda\,\text{constructor}(x_1 : T_1,\, x_2 : T_2\, \ldots)$ just like an ordinary function, by substituting the corresponding values for all free occurrences of $x_1, x_2, \ldots$.

The rule for case statements is:

- Inductive case reduction: if χ has an inductive type, we can replace $(\phi \mid \psi \mid \ldots)(\chi)$ by one of $\phi(\chi), \psi(\chi), \ldots$, depending on which constructor was used to build χ.

For example, if χ is a tree that was built as $\text{node}(l : \text{tree},\, r : \text{tree})$, and if ϕ begins with $\lambda\,\text{node}(l : \text{tree},\, r : \text{tree})$, then we can combine the above two reduction rules to replace $(\phi \mid \psi)(\chi)$ by the body of ϕ with l and r replaced by the left and right children of χ.

> *Adding inductive types doesn't increase the power of first-order logic, in the sense that we can't encode any computations that we couldn't encode before. But inductive types make it much easier and clearer to encode some computations; without them, we'd have to simulate the same ideas using base types and functions. This is possible but less convenient.*

Structural induction

To prove a property for objects of an inductive type M, we can use a generalization of the induction principle called *structural induction*. Let $p(x : M)$ be a predicate. Then:

- Structural induction: consider each constructor c for type M. If c is a base case constructor, we must prove $p(x)$ when x is constructed using c. If c is an inductive constructor, we may assume $p(y)$ for each argument y to c

that has type M; we must then prove $p(x)$ when x is constructed using c. If we can do the above for all constructors, we may conclude $\forall x : M.\ p(x)$.

Example: heaps

A *heap* is a binary tree that stores a number at each node. The numbers are required to satisfy the *heap property*: at every node x with children y, z, the value at x is at least as large as the values at y, z.

We can declare a heap as an inductive type:

$$\textbf{type}\ \text{heap} = \text{leaf}(v : \text{int}) \mid \text{node}(v : \text{int},\ l : \text{heap}, r : \text{heap})$$

We can extract the value at a node by using a case statement to make a function $\text{value}(x)$ that maps $x : \text{heap}$ to int:

$$\text{value} = [(\lambda\, \text{leaf}(v : \text{int}).\ v) \mid (\lambda\, \text{node}(v : \text{int},\ l : \text{heap},\ r : \text{heap}).\ v)]$$

And we can state the heap property:

$$\begin{aligned}
&\forall x : \text{heap}.\ [(\\
&\quad (\lambda\, \text{leaf}(v : \text{int}).\ 0) \mid \\
&\quad (\lambda\, \text{node}(v : \text{int},\ l : \text{heap},\ r : \text{heap}).\ v - \max(\text{value}(l), \text{value}(\text{r}))\\
&)\ x] \geq 0
\end{aligned}$$

Here the function $\max$ of type $(\text{int} \times \text{int}) \to \text{int}$ computes the maximum of two integers:

$$\forall x, y : \text{int}.\ (x \geq y) \to (\max(x, y) = x)$$

$$\forall x, y : \text{int}.\ (x \leq y) \to (\max(x, y) = y)$$

The heap property implies that the largest number in a heap is at the root. So, based on the above assumptions and definitions, we can use recursion to extract the largest number from a heap (without depending on the heap property), and we can use structural induction to prove that this number is always equal to the value at the root.

To extract the maximum element of a heap recursively, we write:

$$\begin{aligned}\mathrm{maxheap} = [&\lambda\,\mathrm{leaf}(v : \mathrm{int}).\ v \mid \\ &\lambda\,\mathrm{node}(v : \mathrm{int},\ l : \mathrm{heap},\ r : \mathrm{heap}). \\ &\quad \max(v, \max(\mathrm{maxheap}(l), \mathrm{maxheap}(r)))]\end{aligned}$$

To prove the max is at the root, we use structural induction with the induction hypothesis

$$\mathrm{value}(x) = \mathrm{maxheap}(x)$$

There is one base case, which is when x is constructed as $v : \mathrm{int}$. In this case, $\mathrm{maxheap}(x)$ and $\mathrm{value}(x)$ both follow their first case. So, they both function-reduce to v, which means that we can use reflexivity of $=$ to show $p(x)$.

There is one inductive case, which is when x is constructed as $(v : \mathrm{int},\ l : \mathrm{heap},\ r : \mathrm{heap})$. In this case we get to assume that l and r satisfy

$$\mathrm{value}(l) = \mathrm{maxheap}(l)$$

$$\mathrm{value}(r) = \mathrm{maxheap}(r)$$

The case statement for $\mathrm{value}(x)$ follows its second case and reduces to v. So, we just need to show that $\mathrm{maxheap}(x)$ also reduces to v.

The case statement for $\mathrm{maxheap}(x)$ follows its second case,

$$\max(v, \max(\mathrm{maxheap}(l), \mathrm{maxheap}(r)))$$

which is equivalent to

$$\max(v, \max(\mathrm{value}(l), \mathrm{value}(r)))$$

by substitution of equals with the induction hypothesis.

From our assumptions (namely the second case of the heap property), we have that x satisfies

$$v \geq \max(\mathrm{value}(l), \mathrm{value}(r))$$

which means (via the definition of max) that

$$\max(v, \max(\mathrm{value}(l), \mathrm{value}(r))) = v$$

We've now shown that $\mathrm{value}(x) = v$ and $\mathrm{maxheap}(x) = v$. So, via reflexivity and substitution of equals, we get

$$\text{value}(x) = \text{maxheap}(x)$$

which is what we needed to prove. So, we conclude

$$\forall x : \text{heap}.\ \text{value}(x) = \text{maxheap}(x)$$

as desired.

Structural induction and plain induction

Structural induction is a strict generalization of induction on the natural numbers. To see why, note that we can define natural numbers as an inductive type:

$$\textbf{type}\ \mathbb{N} = \text{zero}() \mid S(x : \mathbb{N})$$

With this definition, the inference rule for structural induction looks exactly like the inference rule we defined earlier for induction on the natural numbers.

> *As we noted earlier, classical first-order logic can't rule out non-standard numbers — numbers that can't be reached by repeatedly applying the successor function starting from zero. Similarly, for inductive types, first-order logic can't rule out non-standard instances — instances that can't be constructed recursively from base cases. These non-standard instances aren't usually harmful; but if we have to, it is possible to rule them out by moving away from the classical first-order version of our logic.*