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 $\text{first}(s : \text{string})$ and $\text{rest}(s : \text{string})$ that return the first character and all but the first character, respectively.

We'll talk more below about how to define the string *type.*

We define a type state and a predicate $\text{reachable}(s : \text{state})$. We assert $\text{reachable}(s_0, t)$ 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 state s to state s' and consumes a character q, we assert

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

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

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

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

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

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

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 a couple of of its shortcomings: we'll give a more convenient way to talk about computation, and we'll add a way to define new, expressive types called *inductive types*. Both of these additions will pave the way for us to discuss iteration, recursion, and runtime.

Models of computation

There are actually multiple ways to express statements about computation in logic. At some level they are all equivalent, but they can differ greatly in how convenient they are to work with.

The first way is what we did in the previous section: we can define types and predicates that let us work with automata, like finite automata or Turing machines. Unfortunately, there's a reason that there are no popular programming languages based on Turing machines.

The second way is to add extra semantics to the reduction rules that we've already defined. Each reduction represents a step in a computation — for example, calling a function on a given list of arguments, or extracting an element from a given tuple. The main thing we need to add is to define an *order of evaluation*: for example, when calling a function, do we greedily evaluate the arguments first, or do we lazily pass them into the function as-is for later evaluation? Once we define an order of computation, we get a fairly general functional programming language. (A *functional* language is one where the operations don't have side effects like allocating or writing to memory.)

That brings us to the third way, which is the last one we'll discuss: we can define and reason about the state of a model of a computer. For example, our model computer could have a heap where we can allocate memory cells that store different types of values; it could have a stack, a stored program, and a program counter; and it could have input and output channels. The program could be in a high-level language like Python, or a lower-level one like a simplified assembly language. (In the latter case the model is often called a *virtual machine*.) Given this model computer, we define rules that let us update the state: at each update, we execute an instruction from the stored program, and the instruction can read or write memory, update the program counter to jump to a new subroutine, and so forth. This is probably the most common and familiar representation for making logical statements about computation.

For our purposes, we'll mainly use the second way. But we won't give complete low-level details, since those would take up more time and space than we want to spend in this course. Instead we'll give a more informal semantics for computation, with the understanding that it would be possible to expand out the details — that is, we could build on our informal semantics to make a complete language with well-defined inference rules, so that we have a full model of computation within our logic.

> *The idea that logic can encode computation (and vice versa) is an old one. It turns out that there is a very deep connection between these two formalisms: 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****.*

Computation syntax

We'll write programs in a simple programming language, using a notation that is similar to what we've been using so far. As mentioned above, we will wind up with a functional programming language (no assignments or other side effects). All of our computation will be expressed as reduction of terms, with a couple of small exceptions described below.

In more detail, we'll use our language of functions, tuples, etc. We'll pick a *greedy, left-to-right* evaluation order for terms. That is, we simplify the inputs to a reduction as much as possible before applying the reduction, and we work on subexpressions in the same order that they appear in our program. For example, in a function application like $f(a, b)$, we first evaluate f, a, b in that order, then substitute the expressions for a and b into the expression for f.

Since we mostly won't be using propositions, we won't provide a general-purpose evaluation order for them. Instead we'll specify special-purpose rules when they become relevant.

> *To be complete, we would probably want to define rules for* ***shortcut*** *evaluation. For example, for tuple extraction (like* first*) and for case statements on union types, only some of the subexpressions are relevant, and we would want to avoid evaluating irrelevant subexpressions where possible. Instead of defining rules, we'll specifically*

mention shortcutting if it is relevant.

To make it easier to express programs, we'll add two additional syntaxes: named functions and a version of an if-then-else statement. For named functions, we'll write

$$\text{def } f(x)\ \phi$$

to be equivalent to

$$f = \lambda x.\, \phi$$

together with the evaluation strategy that we reduce f to $\lambda x.\, \phi$ when we can (and never go in the opposite direction). Here f is the function name, and ϕ is the function body; ϕ may contain references to f and other function names. For simplicity we require instances of def to occur only in the global scope.

For if-then-else, we'll write

$$p_1 \rightarrow v_1 \mid p_2 \rightarrow v_2 \mid \ldots \mid p_n \rightarrow v_n$$

to mean that we find the first predicate out of $p_1 \ldots p_n$ that is true, and return the value of the corresponding term from $v_1 \ldots v_n$. We require each p_i to reference only current program state (stored values that are reachable from variables in scope), so that it can be evaluated immediately. We require p_n to be just T, so that the overall value of the if-then-else is always well-defined.

The if-then-else syntax is equivalent to defining a new value v with rules

$$p_1 \rightarrow (v = v_1) \quad (p_2 \wedge \neg p_1) \rightarrow (v = v_2) \quad \ldots$$

together with an evaluation strategy that evaluates the guards p_i in order until one returns true, followed by the selected term v_i, and substitutes the evaluated v_i in place of the if-then-else statement.

Note that there are two places where we have allowed propositional expressions into our program — inside function definitions as $f = \lambda x.\, \phi$ and inside if-then-else statements as guards. We have defined an evaluation order for both situations.

Using the above syntax, we can for example write a recursive program to calculate Fibonacci numbers:

```
def $\text{fib}(x)$ [
    $(x \le 1) \to 1$ |
    $T \to \text{fib}(x-1) + \text{fib}(x-2)$
]
```

Using our greedy evaluation order, we would then compute $\text{fib}(5)$ as follows:

$$\begin{aligned}
&\text{fib}(5)\\
&\text{fib}(4) + \text{fib}(3)\\
&\text{fib}(3) + \text{fib}(2) + \text{fib}(2) + \text{fib}(1)\\
&\text{fib}(2) + \text{fib}(1) + \text{fib}(1) + \text{fib}(0) + \text{fib}(1) + \text{fib}(0) + 1\\
&\text{fib}(1) + \text{fib}(0) + 1 + 1 + 1 + 1 + 1 + 1\\
&1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 8
\end{aligned}$$

Note that it will be expensive to evaluate $\text{fib}(x)$ this way, and this is not even close to the best algorithm for computing $\text{fib}(x)$; we'll say more about this observation later.

Other loops

As we saw with the program for fib, our programming language has no trouble with recursion. We can use recursion to simulate all of the familiar types of loops — for, while, etc. But for simplicity we'll add syntax to represent a kind of loop. This includes once-per-iteration updates to loop variables, which we write as something like $i \leftarrow i + 1$. It's important to remember, though, that these are not real assignments: we will instead simulate them with recursion. More specifically, our loop will have four parts: an initialization of the loop variables, a test for completion, an update to the loop variables, and a return value. We'll write it as

```
with $i \leftarrow 1$
while $i \le n$
do $i \leftarrow i + 1$
return $2i^2 - 3$
```

For example, we can write a better program for calculating Fibonacci numbers:

```
def $\mathrm{fib}(x)$ [
    $(x \leq 1) \to 1$ |
    $T \to$
        with $(a \to 1, b \to 1, i \to 2)$
        while $i \leq x$
        do $(a, b, i) \leftarrow (b, a + b, i + 1)$
        return $b$
]
```

The loop above would reduce to the equivalent recursive program

```
def $\mathrm{loop}(a, b, i)$ [
    $(i \leq x) \to \mathrm{loop}(b, a + b, i + 1)$
    $T \to b$
]
$\mathrm{loop}(1, 1, 2)$
```

If we trace out the evaluation of the new version of $\mathrm{fib}(x)$, we can see that it runs much more quickly than our previous version for large values of x.

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). We construct and work with inductive types using recursion or iteration, taking advantage of the computational idioms defined above plus a couple more we will define below. For example, we build big trees by putting together littler trees, and we can traverse these trees with recursive functions like the sum function defined below.

To handle 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 one example, we could define a string as

$$\textbf{type}\ \mathrm{string} = \mathrm{empty}() \mid \mathrm{cat}(f : \mathrm{char}, r : \mathrm{string})$$

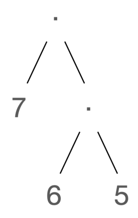
This definition gives two ways to construct a string: either it can be empty (in which case we don't need to provide any arguments), or it can be a character followed with a shorter string (using cat as short for "concatenate", with arguments f for first and r for rest). So the string "hi" would look like

$$\text{cat}(\text{'h'}, \text{cat}(\text{'i'}, \text{empty}()))$$

For another example, we can define a binary tree with two constructors:

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

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 $\text{node}(\text{leaf}(7), \text{node}(\text{leaf}(6), \text{leaf}(5)))$, which looks like this:



In each type definition, the constructors like empty, cat, leaf, and node become new functions. Each one has its own output type: $\text{empty}()$, $\text{cat}(\text{char}, \text{string})$, $\text{leaf}(\text{int})$, and $\text{node}(\text{tree}, \text{tree})$. These new types are defined to be subtypes of string or tree; in fact, string and tree behave like union types, with one case per constructor. The constructor names are arbitrary, but each constructor must be distinct from the others in the same type definition (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.

> *We can use destructuring bind to pull out parts of a value of an inductive type, and then use constructors to include these parts in another value. Naively we might expect to have to do a deep copy in this case, to avoid sharing structure. But since our language is functional, it's perfectly safe for multiple instances to share memory. This can be important, since it could be expensive to do deep copies everywhere. It does however mean that garbage collection or some similar strategy becomes necessary if we want to avoid memory leaks.*

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.

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

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 $\mathrm{leaf}(\mathrm{int})$ or as $\mathrm{node}(\mathrm{tree}, \mathrm{tree})$.

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 $\lambda\,\mathrm{constructor}(x_1 : T_1,\ x_2 : T_2\ \ldots)$ just like an ordinary function, by substituting the corresponding argument 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 both of these rules, our evaluation order is greedy and left-to-right, and we would want to define rules for shortcut evaluations. All of these properties are just like the ones for ordinary function application and union case statements.*

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

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

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:

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

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:

$$\text{def}\max(x, y).\ [(x \geq y) \to x \mid T \to y]$$

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.

In the proof we'll need the following easy-to-derive property of $\max$:

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

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

```
def maxheap($x$) [
    $\lambda\,$leaf($v$ : int). $v$ |
    $\lambda\,$node($v$ : int, $l$ : heap, $r$ : heap).
        $\max(v, \max(\text{maxheap}(l), \text{maxheap}(r)))$
]($x$)
```

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

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

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

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

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

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

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

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

which is equivalent to

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

by substitution of equals with the induction hypothesis.

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

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

which means (via the property of max that we mentioned above) that

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

We've now shown that $\text{value}(x) = v$ and $\text{maxheap}(x) = v$. 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.

Exercise: write a function that removes the max value from the root of a heap, and returns this max value along with an updated heap that contains the remaining values. Prove that the function maintains the heap property.

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. Instead, the rule of induction just tells us that any non-standard numbers must satisfy the same predicates that the standard ones do. 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.