Utrecht University

Logic for Computer Science

09 – Induction

Paige Randall North
(based on Wouter Swierstra's slides)

University of Utrecht

Last time

Relations

This lecture

Induction

Looking ahead

- Next week the mid-term exam is on Monday. The mid-term exam will be **on campus** using Remindo.
- Next week there will be **no quiz** – focus on preparing for the exam instead. We'll cover the material from this lecture in the quiz after Christmas.
- This Thursday there will be a review lecture by Wouter. He will say more about what the mid-term covers and answer any of your questions.
- Regular lectures next week Tuesday (19/12) on *proofs by induction* and *games*.

Equivalence relations

An **equivalence relation** is a relation that is:

- reflexive – $R(x, x)$ for all x.
- symmetric – $R(x, y)$ implies $R(y, x)$
- transitive – $R(x, y)$ and $R(y, z)$ implies $R(x, z)$

The canonical example of such a relation is equality.

Equivalence relations

An **equivalence relation** is a relation that is:

- reflexive – $R(x, x)$ for all x.
- symmetric – $R(x, y)$ implies $R(y, x)$
- transitive – $R(x, y)$ and $R(y, z)$ implies $R(x, z)$

The canonical example of such a relation is equality.

When a relation is an equivalence relation, we often

- write $x \sim y$ for $R(x, y)$
- say x is *equivalent* to y for x is *related* to y.

Equivalence classes

Given an equivalence relation $\sim$ on A, we can define the **equivalence class** of all the elements related to some $a \in A$ as follows:

$$[a]_\sim := \{x \in A \ : \ a \sim x\}$$

This characterizes all the elements equivalent to a under $\sim$.

Equivalence classes

Given an equivalence relation $\sim$ on A, we can define the **equivalence class** of all the elements related to some $a \in A$ as follows:

$$[a]_\sim := \{x \in A \ : \ a \sim x\}$$

This characterizes all the elements equivalent to a under $\sim$.

Now consider *the set* of all the equivalence classes, written $A/\!\sim$:

$$A/\!\sim \ := \{[a]_\sim \ : \ a \in A\}$$

For example: if $\sim$ relates numbers with the same remainder after division by two, the equivalences classes are the sets of even and odd numbers.

Partions

Theorem

Given an equivalence relation $\sim$ on A, the equivalence classes $A/\sim$ form a partition of A.

Intuitively, the equivalence relation divides A into disjoint pieces.

Functions over equivalence classes

To define a function $f : A/{\sim} \to B$ from a set of equivalence classes, we can

- define a function $f' : A \to B$, and then
- check that for all $a \in A$ and $a' \in A$, if $a \sim a'$ then $f(a) = f(a')$.

In other words:

- We check that f' maps *related* inputs to the *same* output.
- Or put differently, f' cannot distinguish between related inputs.
- People often say that f is *invariant* under $\sim$.

Applications

We do this in computer science all the time:

- A compiler is a function on programs (that should not distinguish between the same program using different variable names).
  - (invariance under α-*equivalence*)
- Calculating the surface area of a shape should be independent of *where* the shape is located.
  - (translation and rotation invariance)
- One can represent a set of elements as an array (provided I never observe the *order* of the elements).
  - (invariance under reordering)

Equivalence classes is a mathematical construction which allows us *hide* certain unimportant information.

Example

Suppose we have some class `Car` storing information about a car's make, model, color, etc.

We can define an equivalence relation on `Car` objects easily enough:

$$\texttt{c} \sim \texttt{d} \text{ iff } \texttt{c.colour} = \texttt{d.colour}$$

(Check: Why is is an equivalence relation?)

What kind of functions can we define on the equivalence classes that we get by partitioning all cars by their color?

Example

Does this define a function on equivalence classes?

```
showColor : Car -> String
showColor(c) = toString(c.color)
```

Example

Does this define a function on equivalence classes?

```
showColor : Car -> String
showColor(c) = toString(c.color)
```

Yes! The `showColor` returns the same string for two cars in the same equivalence class: a red Fiat and a red Ferrari will both produce the string “red”.

Example

Does this define a function on equivalence classes?

```
isEV : Car -> Bool
isEV (c) = isElectric(c.motor)
```

Example

Does this define a function on equivalence classes?

```
isEV : Car -> Bool
isEV (c) = isElectric(c.motor)
```

No! A red Tesla and a red Ferrari are in the same equivalence class (they are both red) – yet one will produce `True`; the other will produce `False`.

Example - cars and colours

This example shows that by considering the *equivalence classes* of cars, we limit the information you can use:

- We can observe a car's color;
- But cannot inspect it's make, model, motor, etc.

This is a common pattern in Computer Science, where you want to *hide* certain implementation details.

Equivalence classes gives us the mathematics to do so.

Theorems

We can use this result to turn any *surjection* into a bijection.

Theorems

We can use this result to turn any *surjection* into a bijection.

Given a function $f : A \to B$, define the equivalence relation $\sim_f$ by $x \sim_f x'$ iff $f(x) = f(x')$.

Theorem

Any surjection $f : A \to B$ gives rise to a bijection $\widetilde{f} : A/{\sim_f} \to B$.

Theorems

We can use this result to turn any *surjection* into a bijection.

Given a function $f : A \to B$, define the equivalence relation $\sim_f$ by $x \sim_f x'$ iff $f(x) = f(x')$.

Theorem

Any surjection $f : A \to B$ gives rise to a bijection $\widetilde{f} : A/{\sim_f} \to B$.

Proof. Check that:

- $\sim_f$ is an equivalence relation,
- there is a function $\widetilde{f}\ A/{\sim_f} \to B$, and
- and this function is a bijection.

Theorems

We can use this result to turn any *surjection* into a bijection.

Given a function $f : A \to B$, define the equivalence relation $\sim_f$ by $x \sim_f x'$ iff $f(x) = f(x')$.

Theorem

Any surjection $f : A \to B$ gives rise to a bijection $\widetilde{f} : A/{\sim_f} \to B$.

Proof. Check that:

- $\sim_f$ is an equivalence relation,
- there is a function $\widetilde{f}\ A/{\sim_f} \to B$, and
- and this function is a bijection.

Proof can be found (at the end) of the last set of slides.

Why equivalence classes?

This is a first ‘non-obvious’ example of a mathematical construction that has many applications.

The previous proof relies on bringing together a great deal of material we’ve covered in the previous weeks:

- propositional and predicate logic;
- proof sketches;
- notions of injectivity and surjectivity;
- relations and equivalence classes;
- . . .

Understanding the proof is a good way to stress test your own understanding of this material.

Defining relations

In computer science, we are used to defining functions as formulae:

$$f(x) := x^3 + 17$$

We can study and define these functions without every talking about their graphs.

Defining relations

In computer science, we are used to defining functions as formulae:

$$f(x) := x^3 + 17$$

We can study and define these functions without every talking about their graphs.

In math, relations and functions are usually defined as a subset of $A \times B$, which doesn't give you a 'language' to **define** relations.

Once we cover induction and recursion, we can give a more precise account of how to define relations on infinite sets – and define more interesting relations than the ones we have covered today.

Induction

Recap

Until now we've studied a variety of mathematical tools that we can use to model data, programs and specifications including:

- sets
- functions
- relations

Recap

Until now we've studied a variety of mathematical tools that we can use to model data, programs and specifications including:

- sets
- functions
- relations

But let's take a step back and reflect on **how** to define these things.

Small problems

We can define a **finite** set by enumerating all its elements:

$$\texttt{People} := \{\texttt{Alice}, \texttt{Bob}, \texttt{Carol}\}$$

Small problems

We can define a **finite** set by enumerating all its elements:

$$\texttt{People} := \{\texttt{Alice}, \texttt{Bob}, \texttt{Carol}\}$$

We can define a function on such a finite set by listing all possible cases:

- $\texttt{age}(\texttt{Alice}) = 23$
- $\texttt{age}(\texttt{Bob}) = 21$
- $\texttt{age}(\texttt{Carol}) = 19$

Small problems

We can define a **finite** set by enumerating all its elements:

$$\mathtt{People} := \{\mathtt{Alice}, \mathtt{Bob}, \mathtt{Carol}\}$$

We can define a function on such a finite set by listing all possible cases:

- $\mathtt{age}(\mathtt{Alice}) = 23$
- $\mathtt{age}(\mathtt{Bob}) = 21$
- $\mathtt{age}(\mathtt{Carol}) = 19$

We can define a relation by listing all the relevant pairs:

$$\mathtt{Likes} = \{(\mathtt{Alice}, \mathtt{IceCream}), (\mathtt{Alice}, \mathtt{Toffee}), (\mathtt{Carol}, \mathtt{Toffee})\}$$

Big problems

But what if our data, functions and relations are **infinite**?

We can define the natural numbers as:

$$\mathbb{N} = \{0, 1, 2, \ldots\}$$

We (as humans) can understand this definition perfectly well – but it relies on ‘guessing’ how to fill in the dots.

Question

Why is this definition unsatisfactory?

Criticism

- How could we expect a computer to understand such a definition?
- How can we be sure that the reader ‘guesses’ the right definition? Maybe I meant to define the set of solutions to the equation

$$x \times (x - 1) \times (x - 2) = 0.$$

- The order of elements in a set is not important. Yet this definition implies that the elements should be listed in some particular order.
- How can we determine whether a particular number is in the set or not? The definition doesn’t give us an effective check.
- What about sets where the ‘next’ element is difficult to describe, like the set of all real numbers or the set of all valid C# programs.

We need a better means to describe infinite sets!

Induction

Many infinite sets are described using **induction**.

Each inductive definition consists of three parts:

1. The **base case** that establishes some objects are in the set.
2. The **inductive case** that determines the ways in which elements of the set can be assembled to create new elements that are also in the set.
3. The **extremal clause** that asserts that no other elements are in the set besides the ones asserted in the first two clauses.

(Many definitions only list the first two, leaving the third clause implicit.)

Example – natural numbers

We can give an inductive definition of the natural numbers $\mathbb{N}$ as follows:

- $0 \in \mathbb{N}$
- for any $n \in \mathbb{N}$, the number $(n + 1) \in \mathbb{N}$.
- there are no other elements of $\mathbb{N}$.

Using these clauses, we can show that $3 \in \mathbb{N}$ but $4.5 \notin \mathbb{N}$.

This inductive definition lets us give a **finite** description of an **infinite** set.

Exercise

Give an inductive definition of the even numbers.

Example – power set

Given a set A, we can define the *finite powerset* of A, written $\mathcal{P}(A)$ as follows:

- $\emptyset \in \mathcal{P}(A)$
- if $a \in A$ and $X \in \mathcal{P}(A)$ then $\{a\} \cup X \in \mathcal{P}(A)$
- there are no other elements of $\mathcal{P}(A)$

Let $B = \{1, 2, 3\}$ then from these rules we can conclude that:

- $\emptyset \in \mathcal{P}(B)$
- $\{1\} \cup \emptyset \in \mathcal{P}(B)$ – or more simply $\{1\} \in \mathcal{P}(B)$. Similarly, $\{2\} \in \mathcal{P}(B)$, $\{3\} \in \mathcal{P}(B)$
- Repeating the second rule also gives us that, $\{1, 2\} \in \mathcal{P}(B)$, $\{1, 3\} \in \mathcal{P}(B)$, $\{2, 3\} \in \mathcal{P}(B)$
- Finally, $\{1, 2, 3\} \in \mathcal{P}(B)$.

Objection

Strictly speaking, there is a problem with our definition of the natural numbers:

- for any $n \in \mathbb{N}$, the number $(n + 1) \in \mathbb{N}$.

How is addition defined?

Addition is a binary function on natural numbers – but weren't we trying to define natural numbers in the first place?!

It seems a bit circular to define the natural numbers in terms of an operation on the natural numbers...

Natural numbers revisited

Let's define the natural numbers in the following way.

- $0 \in \mathbb{N}$
- for any $n \in \mathbb{N}$, the number $s(n) \in \mathbb{N}$ (Here we thing of s as being a unary function symbol that stands for 'successor'.)
- There are no other elements of $\mathbb{N}$.

We consider the digit 4 to be a shorthand for $s(s(s(s(0))))$. The Arabic numerals are simply a shorthand for repeatedly adding one using the successor operation.

Later, we'll consider how to define *addition* itself using this definition of natural numbers.

Strings

We can also give an inductive definition of ASCII strings:

- the empty string (denoted ϵ) is a string;
- if c is one of the 256 ASCII characters and s is a string, we can construct a longer string by writing cs (that is, the character c followed by the string s).

Strings

We can also give an inductive definition of ASCII strings:

- the empty string (denoted ϵ) is a string;
- if c is one of the 256 ASCII characters and s is a string, we can construct a longer string by writing cs (that is, the character c followed by the string s).

There is very little that is specific to ASCII in this definition!

Given any set A, we can construct the words of characters over some set A, often written A^* as follows:

- $\epsilon \in A^*$
- for all $a \in A$ an $w \in A^*$, $aw \in A^*$.

Strings

We can also give an inductive definition of ASCII strings:

- the empty string (denoted ϵ) is a string;
- if c is one of the 256 ASCII characters and s is a string, we can construct a longer string by writing cs (that is, the character c followed by the string s).

There is very little that is specific to ASCII in this definition!

Given any set A, we can construct the words of characters over some set A, often written A^* as follows:

- $\epsilon \in A^*$
- for all $a \in A$ an $w \in A^*$, $aw \in A^*$.

Exercise

Define the set of non-empty words (denoted A^+) over a set A.

Examples

Let's try to construct some example inhabitants of the set $\{0, 1\}^*$.

Examples

Let's try to construct some example inhabitants of the set $\{0, 1\}^*$.

- $\epsilon \in \{0, 1\}^*$

Examples

Let's try to construct some example inhabitants of the set $\{0,1\}^*$.

- $\epsilon \in \{0,1\}^*$
- $0 \in \{0,1\}^*$ and $1 \in \{0,1\}^*$

Examples

Let's try to construct some example inhabitants of the set $\{0, 1\}^*$.

- $\epsilon \in \{0, 1\}^*$
- $0 \in \{0, 1\}^*$ and $1 \in \{0, 1\}^*$
- 00, 01, 10, 11 are all also in $\{0, 1\}^*$.

Examples

Let's try to construct some example inhabitants of the set $\{0, 1\}^*$.

- $\epsilon \in \{0, 1\}^*$
- $0 \in \{0, 1\}^*$ and $1 \in \{0, 1\}^*$
- 00, 01, 10, 11 are all also in $\{0, 1\}^*$.
- As are 000, 001, 010, 100, $\ldots$

Better notation, richer structures

Backus-Naur notation

Rather than define such sets using bullet points, the **Backus-Naur Form** (BNF) allows these sets to be described using special notation.

For example, we can define the set of binary words as follows:

$$w ::= \epsilon \mid 0w \mid 1w$$

This says that:

- ϵ is a word
- if w is a binary word, so is $0w$
- if w is a binary word, so is $1w$

Similarly, we can define the natural numbers as:

$$n ::= 0 \mid s(n)$$

Propositional logic

In the previous lectures, we defined the formulae of propositional logic built from some atomic set of formulae P as:

- $\top$ (true) and $\bot$ (false) are formulae;
- every atomic formula P is a formula;
- if p is a formula, then so is $\neg p$;
- if p and q are formulae, then so are $p \wedge q$, $p \vee q$, $p \Rightarrow q$, and $p \Leftrightarrow q$.

Using BNF notation this can be expressed as:

$$p, q ::= \top \mid \bot \mid P \mid \neg p \mid p \wedge q \mid p \vee q \mid p \Rightarrow q \mid p \Leftrightarrow q$$

This makes the structure of propositional logic formulae precise – and we will see how to define functions or relations that manipulate them.

BNF notation

$$p, q ::= \top \mid \bot \mid P \mid \neg p \mid p \wedge q \mid p \vee q \mid p \Rightarrow q \mid p \Leftrightarrow q$$

Note that there is some information left implicit:

- The variable names p and q are propositions.
- Variables with some other name, like P refer to something else.
- We leave implicit that P ranges over the set of atomic propositional formulae.
- There are certain constants, such as $\top$ and $\bot$, that do not refer to some other set like P does.
- This fixes the **structure** of formulae (e.g. conjunction is a binary operation, whereas negation is a unary operation), but does not say anything about their meaning (e.g. how to fill in a truth table).

Programming languages

The BNF notation can also be used to define *programming languages*:

$$
\begin{aligned}
e &::= n \mid x \mid e + e \mid e \times e \mid \dots \\
b &::= \top \mid \bot \mid b_1 \vee b_2 \mid b_1 \wedge b_2 \mid e_1 < e_2 \mid \dots \\
p &::= x := e \\
&\qquad \mid p_1 ; p_2 \\
&\qquad \mid \texttt{if } b \texttt{ then } p_1 \texttt{ else } p_2 \\
&\qquad \mid \texttt{while } b \texttt{ do } p
\end{aligned}
$$

Example: sum

```
i := 0;
s := 0;
while i < n do
    i := i + 1;
    s := s + i
```

Beyond numbers

These examples go to show that there are many different sets that we can define using **induction** and BNF.

There are two more that pop up over and over again: lists and binary trees.

Lists

We can define a data type for *lists of numbers* using the following BNF definition:

$$L ::= [\,] \mid n : L$$

Each list is either:

- equal to the empty list $[\,]$ that has no elements in it;
- or consists of two parts:
    - a first element n stored at the *head* of the list;
    - the remainder (or *tail*) of the list.

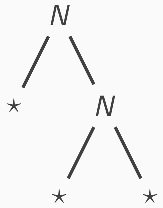
Typically we use some shorthand notation, writing $[1, 2, 3]$ rather than $1 : (2 : (3 : [\,]))$.

Binary trees

We can also store data in other structures, such as *trees*.

The following BNF definition describes the *binary* trees, where each node has two subtrees.

$$t ::= \star \mid N(t_1, t_2)$$



Each node is either:

- a *leaf* $\star$; or
- a *node* $N(t_1, t_2)$ with two subtrees t_1 and t_2.

Inductively defined functions

Inductive definitions

Using BNF we can give a finite description of an infinite set.

But how can we define a *function* that manipulates elements of these sets?

Or define a *relation* between them?

Functions on finite domains

To define a function on a *finite* domain, we typically enumerate all the possible cases:

- $\mathtt{age}(\mathtt{Alice}) = 23$
- $\mathtt{age}(\mathtt{Bob}) = 21$
- $\mathtt{age}(\mathtt{Carol}) = 19$

Such a case analysis, however, does not work if we want to define a function on *infinitely many* inputs. . .

Example: factorial function

The **factorial function**, often written as $n!$ is a function $\mathbb{N} \to \mathbb{N}$.

Intuitively, it is defined as follows:

$$n! := 1 \times 2 \times 3 \times \ldots \times n$$

But this is not a very formal definition!

Once again, we're expecting our reader to fill in the dots.

Example: factorial by cases

If we try to define all possible cases, we'll need infinitely many cases:

$$\begin{aligned}
0! &= 1 \\
1! &= 1 \times 1 \\
2! &= 1 \times 2 = 2 \\
3! &= 1 \times 2 \times 3 = 6 \\
4! &= 1 \times 2 \times 3 \times 4 = 24 \\
\ldots
\end{aligned}$$

Once again, we need a new way to *define* a function.

Example: factorial inductively

Previously, we defined the set of all natural numbers as:

- 0 is a natural number;
- if n is a natural number, so is $s(n)$.

Let's try to follow the same structure to define the factorial function:

- the factorial of 0 is 1;
- if the factorial of n is k, the factorial of $s(n)$ is $s(n) \times k$.

Example: factorial inductively

Previously, we defined the set of all natural numbers as:

- 0 is a natural number;
- if n is a natural number, so is $s(n)$.

Let's try to follow the same structure to define the factorial function:

- the factorial of 0 is 1;
- if the factorial of n is k, the factorial of $s(n)$ is $s(n) \times k$.

Or we might write:

$$
\begin{aligned}
0! &:= 1 \\
(s(n))! &:= s(n) \times n!
\end{aligned}
$$

Example: factorial inductively

$$0! := 1$$

$$(s(n))! := s(n) \times n!$$

Here we have defined the *factorial* function *by induction on its input*.

To successfully do so we need:

- to say what the factorial of 0 is;
- to say how to define the factorial of $s(n)$, assuming we already know what the factorial of n is.

This recipe works for different functions over natural numbers.

Similar inductive definitions work over other inductively defined sets, such as lists, trees, or the formulae of propositional logic.

Example: Fibonacci numbers

The **Fibonacci numbers** are an infinite sequence of numbers:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

We can always compute the next Fibonacci number by adding the previous two together.

Example: Fibonacci numbers

The **Fibonacci numbers** are an infinite sequence of numbers:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

We can always compute the next Fibonacci number by adding the previous two together.

We can this series using induction as follows:

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

Addition using induction

This inductive definitions can be used to define all of mathematics from the ground up.

Example

Give an inductive definition of a function

$$\texttt{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

that formally defines the addition of two natural numbers.

Addition using induction

This inductive definitions can be used to define all of mathematics from the ground up.

Example

Give an inductive definition of a function

$$\mathtt{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

that formally defines the addition of two natural numbers.

$$\mathtt{add}(0, n) := n$$

Addition using induction

This inductive definitions can be used to define all of mathematics from the ground up.

Example

Give an inductive definition of a function

$$\mathtt{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

that formally defines the addition of two natural numbers.

$$\mathtt{add}(0, n) := n$$
$$\mathtt{add}(s(k), n) = s(\mathtt{add}(k, n))$$

Addition using induction

This inductive definitions can be used to define all of mathematics from the ground up.

Example

Give an inductive definition of a function

$$\texttt{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

that formally defines the addition of two natural numbers.

$$\begin{aligned} \texttt{add}(0, n) &:= n \\ \texttt{add}(s(k), n) &= s(\texttt{add}(k, n)) \end{aligned}$$

Check with some examples!

Peano arithmetic

In this fashion, we can define multiplication, exponentiation, and all other familiar arithmetic operations.

By doing so, we can formalize all of primary school mathematics and basic algebra.

This version of the natural numbers are sometimes referred to as the *Peano numbers*, named after the Italian mathematician, linguist and logician Giuseppe Peano (1858-1932) that proposed them.

Beyond natural numbers

We can also find inductive definitions of other functions defined over more interesting sets than natural numbers:

- length of a word
- height of a binary tree
- ...

Example: length of a word

For any set A, we can define the words over the alphabet A in BNF as follows:

$$w ::= \epsilon \mid aw$$

For example, by taking $A = \{0, 1\}$ we get all the binary words.

We can define the *length* of any word as follows:

Example: length of a word

For any set A, we can define the words over the alphabet A in BNF as follows:

$$w ::= \epsilon \mid aw$$

For example, by taking $A = \{0, 1\}$ we get all the binary words.

We can define the *length* of any word as follows:

$$\mathit{length}(\epsilon) = 0$$

Example: length of a word

For any set A, we can define the words over the alphabet A in BNF as follows:

$$w ::= \epsilon \mid aw$$

For example, by taking $A = \{0, 1\}$ we get all the binary words.

We can define the *length* of any word as follows:

$$\begin{aligned} \mathit{length}(\epsilon) &= 0 \\ \mathit{length}(aw) &= s(\mathit{length}(w)) \end{aligned}$$

Example: length of a word

For any set A, we can define the words over the alphabet A in BNF as follows:

$$w ::= \epsilon \mid aw$$

For example, by taking $A = \{0, 1\}$ we get all the binary words.

We can define the *length* of any word as follows:

$$\begin{aligned} \mathit{length}(\epsilon) &= 0 \\ \mathit{length}(aw) &= s(\mathit{length}(w)) \end{aligned}$$

Check with some examples!

Propositional logic

We can use the same techniques to define functions over the formulae of propositional logic.

Question

How can we write a function that given an arbitrary formula p in propositional logic, computes the number of rows in the truth table for p?

Atomic propositions

For example, the function $fv(p)$ computes the set of all the atomic propositional formulae mentioned in p.

$$\begin{aligned} fv(\top) &= \emptyset \\ fv(\bot) &= \emptyset \\ fv(\neg p) &= fv(p) \\ fv(p \wedge q) &= fv(p) \cup fv(q) \\ fv(p \vee q) &= fv(p) \cup fv(q) \\ fv(P) &= \{P\} \end{aligned}$$

For any formula in propositional logic p, the truth table for p will have $2^{|fv(p)|}$ rows.

Recursion

Beware. . .

Consider the following function $f : \mathbb{N} \to \mathbb{N}$:

$$\begin{aligned} f(0) &= 0 \\ f(n) &= f(n+1) \end{aligned}$$

What is the value of $f(1)$?

Beware...

Consider the following function $f : \mathbb{N} \to \mathbb{N}$:

$$
\begin{aligned}
f(0) &= 0 \\
f(n) &= f(n+1)
\end{aligned}
$$

What is the value of $f(1)$?

This function does not terminate on non-zero inputs!

$f(1) = f(1+1) = f(2) = f(2+1) = f(3) = \ldots$

Inductive definitions

The definitions that we have seen so far use *structural induction*:

- We define a set:
  - by its base cases;
  - by its inductive cases;
- We define a function by giving:
  - its values for the base cases;
  - describing how to compute the value for an inductive case in terms of the results of the ‘smaller subexpressions’ (e.g. the rest of the word, the tail of the list, or both subtrees).

The structure of our function definitions follows the structure of our inductively defined set.

For any input, we can compute the result of applying our function by applying the inductive step a **finite** number of times.

Recursive definitions

Alternatively, we can consider definitions using *recursion*.

We define a function by giving:

- its values for the base cases;
- describing how to compute the value for inductive cases by calling the function we are defining on *any* inputs;

This is sometimes called *general recursion* to distinguish it from the ‘safe’ version of recursion that we’ve seen so far.

Example: recursion

$$f(n) = \begin{cases} 1 & \text{when } n \leq 1 \\ f(n/2) & \text{when } n \text{ is even} \\ f(3n+1) & \text{when } n \text{ is odd} \end{cases}$$

Examples:

$$f(1) = 1$$
$$f(2) = f(1) = 1$$
$$f(3) = f(10) = f(5) = f(16) = f(8) = f(4) = f(2) = f(1) = 1$$
$$f(4) = f(2) = f(1) = 1$$
$$f(5) = f(16) = f(8) = f(4) = f(2) = f(1) = 1$$
$$\ldots$$

Recursion

$$f(n) = \begin{cases} 1 & \text{when } n \leq 1 \\ f(n/2) & \text{when } n \text{ is even} \\ f(3n+1) & \text{when } n \text{ is odd} \end{cases}$$

Question:

Does f always terminate?

Recursion

$$f(n) = \begin{cases} 1 & \text{when } n \leq 1 \\ f(n/2) & \text{when } n \text{ is even} \\ f(3n+1) & \text{when } n \text{ is odd} \end{cases}$$

Question:

Does f always terminate?

The answer to this question is unknown!

Computers have checked that the f terminates for all numbers up to 5,764,000,000,000,000, there is no proof that f terminates for all inputs.

This is sometimes referred to as the **Collatz conjecture**.

McCarthy's f91 function

$$f_{91}(n) = \begin{cases} n - 10 & \text{when } n > 100 \\ f_{91}(f_{91}(n+11)) & \text{when } n \leq 100 \end{cases}$$

McCarthy's f_{91} function is another example of a function whose behavior is not at all obvious at first.

Clearly it terminates for inputs greater than 100.

But it turns out that even for inputs smaller than 100, it terminates and always returns 91.

Inductive definitions versus recursive definitions

Inductively defined functions:

- closely follow the inductive structure of its domain;
- may only make recursive calls to the structurally smaller values;
- are guaranteed to terminate and produce a value;

Recursively defined functions:

- may make arbitrary recursive calls – making them strictly more general than just induction;
- may not terminate. . .

Inductive definitions versus recursive definitions

Inductively defined functions:

- closely follow the inductive structure of its domain;
- may only make recursive calls to the structurally smaller values;
- are guaranteed to terminate and produce a value;

Recursively defined functions:

- may make arbitrary recursive calls – making them strictly more general than just induction;
- may not terminate. . .

Oftentimes induction should suffice to define most ‘sensible’ functions – but some definitions require general recursion.

In that case, you need to use more advanced proof techniques to show that a function defined using general recursion is valid and guaranteed to terminate.

Inductively defined relations

So far we have seen inductively defined sets and inductively defined functions.

But can we use these same techniques to define *relations* inductively?

Inductively defined relations

So far we have seen inductively defined sets and inductively defined functions.

But can we use these same techniques to define *relations* inductively?

Yes! This turns out to be the key technique used to define complex relations in Computer Science, such as:

- the semantics of a programming language;
- the type system of a programming language;
- the syntax of a programming language;
- the scoping rules of a programming language;
- the relation defining what constitutes a valid proof;
- ...

Example: less than

We have seen that all natural numbers can be defined as:

- $0 \in \mathbb{N}$
- for any $n \in \mathbb{N}$, we have $s(n) \in \mathbb{N}$

We can define the $\leq$ *relation* between natural numbers using the following rules:

- for all $n \in \mathbb{N}$, $0 \leq n$;
- if $n \leq m$, then $s(n) \leq s(m)$

Example: less than

We have seen that all natural numbers can be defined as:

- $0 \in \mathbb{N}$
- for any $n \in \mathbb{N}$, we have $s(n) \in \mathbb{N}$

We can define the $\leq$ *relation* between natural numbers using the following rules:

- for all $n \in \mathbb{N}$, $0 \leq n$;
- if $n \leq m$, then $s(n) \leq s(m)$

Check that $1 \leq 2$.

Exercise

Give a predicate (i.e., relation with one argument) that characterizes the *sorted* lists of numbers.

Exercise

Give a predicate (i.e., relation with one argument) that characterizes the *sorted* lists of numbers.

- the empty list is sorted;
- a list with one element, $x : [\,]$, is always sorted;
- a list with at least two elements, $x : y : L$ is sorted, provided $x \leq y$ and $y : L$ is also sorted.

Exercise

Give a predicate (i.e., relation with one argument) that characterizes the *sorted* lists of numbers.

- the empty list is sorted;
- a list with one element, $x : [\,]$, is always sorted;
- a list with at least two elements, $x : y : L$ is sorted, provided $x \leq y$ and $y : L$ is also sorted.

In this style we can define arbitrary relations or properties of (inductively defined) sets *precisely* and *unambiguously*.

We'll introduce better notation for inductively defined relations after the Christmas break.

Recap

- Induction is a powerful mathematical technique that can be used to give a *finite* description of an *infinite* set.
- We can also use induction to define functions and relations over inductively defined sets.

Recap

- Induction is a powerful mathematical technique that can be used to give a *finite* description of an *infinite* set.
- We can also use induction to define functions and relations over inductively defined sets.

But how can we *prove* properties of such inductively defined functions and relations?

Material

- Modelling Computing Systems Chapter 8