Teoria da Computação

(Theoretical Computer Science)
Licenciatura em Engenharia Informática
Lecture Notes 2011-2012

Luis Caires

version of October 23, 2011

1 Review of Set Theory, Modeling with Sets

The basic goal of this chapter is to help you learn how to:

- model data spaces and data structures using basic Set Theory
- specify properties of states of a computing system and of elements within a data structures using Logic

1.1 Basic Set Theory

1. Sets, Everything is a Set and ZFC

   Set theory was invented to provide a foundation to model ALL mathematical concepts. In turn mathematical concepts can be used to model most concepts of scientific and technological disciplines. Informatics and computer science are not an exception. It turns out that set theory and mathematical logic are particularly convenient tools to model concepts in informatics and computer science.

   Set theory and logic play for informatics the same basic role as mathematical analysis (calculus) plays for disciplines such as physics or electronic engineering.

   We will base our presentation on ZFC (Zermelo-Fraenkel-Cantor) Set Theory, due to these three famous mathematicians. Set theory was also developed by pioneers of computer science, for example, John Von Neumann.

Set theory is based on the idea that "Everything is a set". Actually, this means "Everything can be modeled by a set". ZFC models things such as boolean values, natural numbers, relations, functions, databases, and even algorithms, just based on the fundamental notion of set.

2. Emptyset

   The empty set is the "simplest" set we may think of. It is the set without elements. It is represented $\emptyset$.

3. Membership

   The fundamental form of statement in set theory is

   $$x \in y$$

   which means "x is a member of y", or "x belongs to y".

4. Extensionality

   The "Extensionality Principle" of set theory means that sets are determined uniquely by their elements. If two sets (finite or infinite) have exactly the same elements, then they are actually the same set. For example, we may think of two Java vectors with exactly the same elements, without being the same vector. This is not the case with sets.

   Practically, if we want to check if two sets A and B are actually the same set, if is enough to check that every element of A also belongs to B, and that every element of B also belongs to A.

   $$A = B \Leftrightarrow (\forall x. x \in A \Leftrightarrow x \in B)$$

   Extensionality also implies that there is just one empty set.

5. Subset

   A set x is a subset of a set y if all elements of x belong to y. Formally, we have

   $$A \subseteq B \Leftrightarrow \forall x.(x \in A \Rightarrow x \in B)$$

   Note that $A \subseteq A$ for all sets A, and $\emptyset \subseteq A$ for all sets A. Sometimes we use $A \subset B$ to say that A is a strict subset of B. A strict subset of a set B is a subset that is not the trivial subset B.

   $$A \subset B \Leftrightarrow (A \subseteq B) \wedge A \neq B$$

6. Enumeration

   We can define sets in various ways.

   The simplest way is by exhaustively enumerating all the elements in the set you want to specify

$$
\begin{array}{lcl}
BOOL & \triangleq & \{FALSE, TRUE\} \\
DWARFS & \triangleq & \{\text{“Sneezy”}, \text{“Sleepy”}, \text{“Dopey”}, \text{“Doc”}, \text{“Happy”}, \\
 & & \text{“Bashful”}, \text{“Grumpy”}\} \\
LAMPSTATES & \triangleq & \{ON, OFF\}
\end{array}
$$

   Obviously, this only works for specifying finite sets.

   When we define a set by enumerating its elements, the order or presentation does not matter! So, the following enumerations define the same set:

$$
\begin{array}{c}
\{1, 2, 3\} \\
\{2, 1, 3\} \\
\{3, 1, 2\}
\end{array}
$$

7. Sets, Sets of Sets, Sets of Sets of Sets, ...

   An set can also be an element of another set, and so on. This is useful to describe structured entities, with several components

$$STACK \triangleq \{0, \{2, \{3\}\}\}$$

$$BOOLS \triangleq \{\emptyset, \{TRUE\}, \{FALSE\}, BOOL\}$$

8. Comprehension

   We may define a new set using a logical property to select the elements we want to collect. For example

   The set of even natural numbers:

$$EVEN \triangleq \{n \in NAT \mid n\%2 = 0\}$$

   The set of non empty sets:

$$NOTEMPTY \triangleq \{s \mid s \neq \emptyset\}$$

   The general form of the “naive” comprehension principle allows us to define a new set given any property P expressed in the logic of set theory.

$$\{x \mid P(x)\}$$

The logic of set theory is essentially first-order logic enriched with several constants and operators that talk about sets, for example, the empty set, the membership relation, equality, etc, etc.

9. Russell's Paradox

   In 1901 Bertrand Russell discovered an inconsistency of Cantor-Frege set theory, by considering the set

   $$R \triangleq \{x \mid x \notin x\}$$

   Intuitively (so to speak), R is the set of all sets that are not members of themselves. Being not a member of itself is a property that make sense, in principle. We can think of many sets that enjoy this property, for example, the empty set is not a member of itself. The set of boolean values is not itself a boolean value.

   Since there are so many examples of sets that are not members of themselves, the set R as defined above, if it exists, must be not empty! We may even naively think that R contains all the sets that exists, since perhaps no set can be a member of itself.

   But a paradox (or inconsistency) arises! Consider the meaning of the proposition

   $$R \in R$$

   By definition of the "set" R, $R \in R$ means that $R \notin R$.

   Likewise, if we assume $R \notin R$, then it cannot be the case that $R \notin R$. So $R \notin R$ implies $R \in R$.

   Even if surprised at first, we must conclude that, according to the definition of R, we have

   $$R \in R \text{ if and only if } R \notin R$$

   which is obviously an absurd.

   Since we arrived to an absurd statement only by following the basic rules of logic and the definition of R, Russell concluded, rightly, that an expression like $\{x \mid x \notin x\}$ must be meaningless, and cannot be used to define a set. Such meaningless expressions should not be accepted by the language of set theory.

10. Separation

    To avoid confusions like Russell's paradox, we will always use Comprehension in a refined form, using the Separation principle of ZFC.

The general idea of the separation principle is that we may define a new set given any property P expressed in the logic of set theory, to select elements from some **already well defined** set S.

$$\{x \in S \mid P(x)\}$$

So, according to this principle, we have the right to write

$$\{n \in NAT \mid n\%2 = 0\}$$

a well defined set, but not an expression such as $\{s \mid s \neq \emptyset\}$.

Be careful to always use the separation principle when defining sets by comprehension in this course!

11. Union

Besides Enumeration and Separation, we may define sets using the Union operation

$$A \cup B$$

Intuitively $A \cup B$ denotes the set that contains exactly the elements in A and B.

$$\forall x.(x \in A \cup B) \Leftrightarrow (x \in A) \vee (x \in B)$$

Given a set of sets S we also define the union $\bigcup S$ to mean the union of all sets which are elements of S. More precisely, we have

$$\forall x.(x \in \bigcup S) \Leftrightarrow \exists y.(y \in S \wedge x \in y)$$

12. Intersection / Disjointness

We may define sets using the Intersection operation

$$A \cap B$$

Intuitively $A \cap B$ denotes the set that contains exactly the elements that belong both to A and to B.

$$\forall x.(x \in A \cap B) \Leftrightarrow (x \in A) \wedge (x \in B)$$

We may also see that

$$A \cap B = \{x \in A \mid x \in B\}$$

Given a set of sets S we also define the intersection $\bigcap S$ to mean the intersection of all sets which are elements of S. More precisely, we have

$$\forall x.(x \in \bigcap S) \Leftrightarrow \forall y.(y \in S \Rightarrow x \in y)$$

Two sets A and B are said to be disjoint, in symbols $A\#B$, if they do not contain any common member. We have

$$A\#B \Leftrightarrow (A \cap B) = \emptyset$$

We say that a collection S of sets is pairwise disjoint if all pairs of sets in the collection are disjoint. More precisely

$$\#S \Leftrightarrow \forall x.\forall y.(x \in S \land y \in S \land x \neq y \Rightarrow x\#y)$$

13. Relative Complement

    Given a sets A and B, the relative complement $A \setminus B$ denotes the set of all elements of A that do not belong to B. Formally

    $$A \setminus B = \{x \in A \mid x \notin B\}$$

    The "absolute complement" of a set A, written $\overline{A}$ is not definable in ZFC, due to the Russell paradox.

14. Pairs

    For structuring information we need some kind of construction to aggregate data. The simplest one is the pair. We may form e.g., a pair consisting of a team and the size of the team.

    $$daltons \triangleq (\{\text{“}jack\text{”}, \text{“}joe\text{”}, \text{“}averell\text{”}, \text{“}william\text{”}\}, 4)$$

    This corresponds to the well known notion of **ordered pair**. In set theory, everything is a set, and in fact an ordered pair such as the one above may be encoded in a set, using the scheme

    $$(x, y) \triangleq \{x, \{x, y\}\}$$

    This encoding of pairs is a variant of one Kuratowski proposed in 1921.

    In practice, we will simply use the standard notation (x, y) to represent ordered pairs.

15. Products

    The product of to sets A and B, written $A \times B$ is the set of all ordered pairs whose first element belongs to A and the second element belongs to B.

    We have

    $$\forall x.(x \in A \times B) \Leftrightarrow \exists a.\exists b.(a \in A \wedge b \in B \wedge x = (a, b))$$

    This operation is also called the "cartesian" product. The name "cartesian" derives from the name of René Descartes, the mathematician-philosoper that invented the related concept of cartesian plane, where one conceive points with two coordinates (x, y) (even if it is best known by his famous punchline "I think therefore I am" :-).

16. Fixed Sequences and n-tuples.

    We may represent tuples of more than 2 elements by iterating the product. For example $STRING \times NAT \times STRING$ denotes the set of all triples (a, b, c) where $a \in STRING$, $b \in NAT$ and $c \in STRING$.

    This idea of forming sets of tuples of any fixed arbitrary length works by considering the operation $A \times B$ to be right associative, so $A \times B \times C$ is actually an abbreviation of $A \times (B \times C)$.

    In the same way a triple such as (a, b, c) is actually an abbreviation of a pair $(a, (b, c))$.

    So we can say, for example, that the first component of (a, b, c) is a and the second component of (a, b, c) is (b, c).

    Note however that a sequence such as $((a, b), c)$ is different from the sequence (a, b, c). The first is a sequence of two elements, namely the pair (a, b) and c, while the second sequence contains three elements, a, b and c.

    This reasoning applies to sequences of elements of arbitrary finite length.

17. Relations

    A (binary) relation between elements of a set A and elements of a set B is modeled as a subset of the product $A \times B$. For example, the relation $SAMEPAR$ that holds between two natural numbers if and only if they have the same parity (odd or even) is defined as follows

    $$SAMEPAR \triangleq \{(x, y) \in NAT \times NAT \mid x\%2 = y\%2\}$$

For example, $(2, 8) \in SAMEPAR$ and $(9, 1) \in SAMEPAR$ but $(191, 256) \notin SAMEPAR$.

When R is supposed to denote a relation, we write $a\,R\,b$ for $(a, b) \in R$, to make it more readable. For example, we may write $2\ SAMEPAR\ 8$.

Here some other examples of binary relations:

$$x\ FATHER\_OF\ y$$

$$n\ ANCESTOR\_OF\ y$$

$$n\ LINKED\_TO\ y$$

We can also define relations between more than 2 elements. For that, we just iterate the constructions above, using products and n-tuples. For example, a phone list may be seen as a relation

$$PHONELIST \subset FIRSTNAME \times LASTNAME \times PHONENUM$$

where we may set $FIRSTNAME \triangleq STRING$, $LASTNAME \triangleq STRING$ and $PHONENUM \triangleq NAT$. For example, we may consider

$$(\text{“Luis”}, \text{“Caires”}, 218402825) \in PHONELIST$$

Relations are an extremely important concept in informatics and computer science. For example, it is pervasive in databases theory and practice, which are based in the so called relational data model, invented by Edgar Codd in 1970. Codd won the 1981 ACM Turing Award for this key contribution to Informatics. The relational model is the basis of most modern database systems, which use the query language SQL. You will learn more about this in the Databases course.

18. PowerSet

We often need to define the set of all subsets of a given set. For example, we may want to consider a specific phonelist, as defined above. To what set does such phonelist belong? Well, a single phonelist is a set of triples (each one representing a record) where each triple belongs to the set

$$FIRSTNAME \times LASTNAME \times PHONENUM$$

The set of all sets of records of these kind is denoted by the powerset

$$\wp(FIRSTNAME \times\ LASTNAME \times PHONENUM)$$

In general, for any sets A and S we have that

$$A \in \wp(S) \Leftrightarrow A \subseteq S$$

19. Functions

A function is modeled in set theory just as a special kind of relation, a relation between arguments and the corresponding results. Since a function cannot give two different results for the same argument, we impose the following condition for a binary relation R to be considered a function

$$\mathit{function}(R) \triangleq \forall (x, y) \in R, \forall (x', y') \in R \,.\, (x = x') \Rightarrow (y = y')$$

This means that if F is a function such that $(\text{“}\mathit{luis}\text{”}, a) \in F$ and $(\text{“}\mathit{luis}\text{”}, b) \in F$ then $a = b$, for example $a = b = 45$. There cannot be two different pairs with the same first component!

We may think of F as the AGE function that assigns to a person its (unique) age.

Since the result b of a function relation F is unique for any given argument, we denote such result by $F(a)$ where a is the first element of the pair $(a, b) \in F$. In the example above, we have, say $F(\text{“}\mathit{luis}\text{”}) = 45$.

So, note that, in the end, a function in set theory is nothing but a set of ordered pairs!

To highlight the use of ordered pairs in the context of functions, we also use the following alternative notation for ordered pairs

$$x \mapsto y \triangleq (x, y)$$

The notation $x \mapsto y$ reads “x is mapped to y” (“x é aplicado em y”).

Given a function as a set (of ordered pairs) we also call such set (of ordered pairs) the **extension** of the function.

For example, the extension of the NOT function on booleans may be represented by:

$$NOT \triangleq \{TRUE \mapsto FALSE, FALSE \mapsto TRUE\}$$

Then, we have $NOT(TRUE) = FALSE$, and $(FALSE, TRUE) \in NOT$.

The set of all subsets of $A \times B$ which are functions is denoted by

$$A \rightarrow B$$

In other words,

$$A \rightarrow B \triangleq \{R \in \wp(A \times B) \mid \mathit{function}(R)\}$$

We may then write, as usual

$$NOT \in BOOL \to BOOL$$

$F \in A \to B$ means that F is a function that sends elements of A into elements of B.

The set A (in $A \to B$) is called the **domain** of the function F, and B the **codomain** of the function F.

There are several ways of defining functions in set theory. An convenient way we will often use is to follow the pattern

$$F \triangleq \{x \mapsto y \in D \times C \mid P(x, y)\}$$

where $P(x, y)$ is a logical condition between the argument x and the result y, D is the domain and C is the codomain. For example,

$$DOUBLE \triangleq \{x \mapsto y \in NAT \times NAT \mid y = 2 \times x\}$$

Then $DOUBLE(2) = 4$, etc...

20. Identity Function

    For any set A there is the identity function on A, that maps each $e \in A$ into itself. The identity on A is noted Id_A. We have

    $$Id_A = \{a \mapsto b \in A \times A \mid a = b\}$$

    so that $Id_A(a) = a$ for all $a \in A$.

21. Projections

    Projections are useful functions that may be used to select elements from pairs and n-tuples.

    Given any product $A \times B$ we define the functions

    $$\pi_1 \triangleq \{((a, b) \mapsto a) \in (A \times B) \times A \mid (a, b) \in A \times B\}$$

    $$\pi_2 \triangleq \{((a, b) \mapsto b) \in (A \times B) \times B \mid (a, b) \in A \times B\}$$

    You may check that for the functions π_1 and π_2 just defined we have

    $$\pi_1 \in (A \times B) \to A$$

    $$\pi_2 \in (A \times B) \to B$$

    For example, $\pi_1((\text{“}luis\text{”}, 45)) = \text{“}luis\text{”}$, and $\pi_2((\text{“}luis\text{”}, 45)) = 45$.

    Projections generalize to n-tuples, for example, we may define the projections π_3, π_4, etc, which operate on triples, 4-tuples, etc.

1.2 Solved modeling problems

1. Model the following system with a structure.

   A lamp with two states `ON` and `OFF`.

   (a) Model the set of states of a lamp with a set $SLAMP$.

   (b) Define a function in $SLAMP \to SLAMP$ that models the "turn on" operation.

   (c) Define a function in $SLAMP \to SLAMP$ that models the "turn off" operation.

   (d) Define a function in $SLAMP \to BOOL$ that returns the current state of the lamp.

   Solution The set of states:

   $$SLAMP = \{0, 1\}$$

   The function of (b)

   $$turn\_on \triangleq \{0 \mapsto 1, 1 \mapsto 1\}$$

   The function of (c)

   $$turn\_off \triangleq \{0 \mapsto 0, 1 \mapsto 0\}$$

   The function of (d)

   $$status \triangleq \{0 \mapsto FALSE, 1 \mapsto TRUE\}$$

   The structure modeling the system:

   $$LAMP \triangleq (SLAMP, turn\_on, turn\_off, status)$$

2. Model the following system with a structure.

   A counter keeps the count of cars inside a tunnel by keeping track if cars entering the tunnel and cars exiting the tunnel.

   (a) Model the set of states of a counter with a set $SCOUNTER$.

   (b) Define a function in $SCOUNTER \to SCOUNTER$ that models the "car enter" operation.

   (c) Define a partial function in $SCOUNTER \to SCOUNTER$ that models the "car exit" operation.

(d) Define a function in $SCOUNTER \to NAT$ that yields the number of cars currently inside the tunnel.

Solution The set of states:

$$SCOUNTER \triangleq NAT$$

The function of (b)

$$car\_enter \triangleq \{n \mapsto m \in NAT \times NAT \mid m = n + 1\}$$

The function of (c)

$$car\_exit \triangleq \{n \mapsto m \in NAT \times NAT \mid n = m + 1\}$$

The function of (d)

$$cars\_inside \triangleq id_{NAT}$$

The structure modeling the system:

$$COUNTER \triangleq (SCOUNTER, car\_enter, car\_exit, cars\_inside)$$

3. Model the following data with sets

   (a) The set of all bank accounts, where each bank account includes the owner name, the account number, and the balance.

   (b) Define a function $JOIN$ that given a set of bank accounts B without repeated account numbers, and two account numbers in B, yields a set of bank accounts identical to the given one, except that the two given accounts are merged in a new account, under the number of (and owner of) smallest account number.

   (c) To what set belongs the function $JOIN$?

Solution We may first define the sets, just for convenience,

$$\begin{array}{lcl} NAME & \triangleq & STRING \\ ACCNUM & \triangleq & NAT \\ AMOUNT & \triangleq & NAT \end{array}$$

(a) The set of all bank accounts

$$ACC \triangleq NAME \times ACCNUM \times AMOUNT$$

An example of a bank account

$$(\text{“luis”}, 1024, 80000000000)$$

We have $(\text{“luis”}, 1024, 80000000000) \in ACC$

(b) Any set of bank accounts B is a subset of ACC, in other words, a member of $\wp(ACC)$.

For any set $B \in \wp(ACC)$ and account numbers n_1 and n_2 in B, we define the set

$$
\begin{aligned}
&\mathit{merge}(B, n_1, n_2) \\
&\triangleq \\
&\{c \in B \mid \pi_2(c) \neq n_1 \wedge \pi_2(c) \neq n_2\} \\
&\cup \\
&\{(o, n, b) \in ACC \mid \\
&\qquad n = \mathit{min}(n_1, n_2) \wedge \exists b_1.\exists b_2.(o, n_1, b_1) \in B \wedge (o, n_2, b_2) \in B \wedge b = b_1 + b_2\}
\end{aligned}
$$

The first part of the union contains the accounts in B that are not the accounts with numbers n_1 or n_2.

The second part of the union contains the "joined" account.

The function $JOIN$ can then be defined

$$JOIN \triangleq \{(S, n_1, n_2) \mapsto M \mid M = \mathit{merge}(S, n_1, n_2)\}$$

(c) We have

$$JOIN \in (\wp(ACC) \times ACCNUM \times ACCNUM) \to \wp(ACC)$$

1.3 Inductive Definitions

We have discussed several ways to define sets, for example, by enumeration, by comprehension, and by applying set operations to previously defined sets.

Another fundamental way of defining sets, particularly useful in informatics and computer science, is the so-called **inductive definition**.

Induction is an extremely powerful technique, and plays in set theory a role similar to the one recursion plays in programming (this remark is only for those of you that already know what is a recursive function or recursive procedure in a programming language).

Using induction, we define sets using an incremental construction method, by adding in stages to previously built stages, as if we were building skyscrapers from their foundations.

Actually, the basic idea is quite simple.

First, we enumerate a (finite) set of basic elements that must belong to the set we want to define. We can think of these basic elements as some kind of "seeds".

You may imagine the "seeds" as being the "basic" elements of the set. This elements will be created in stage 0.

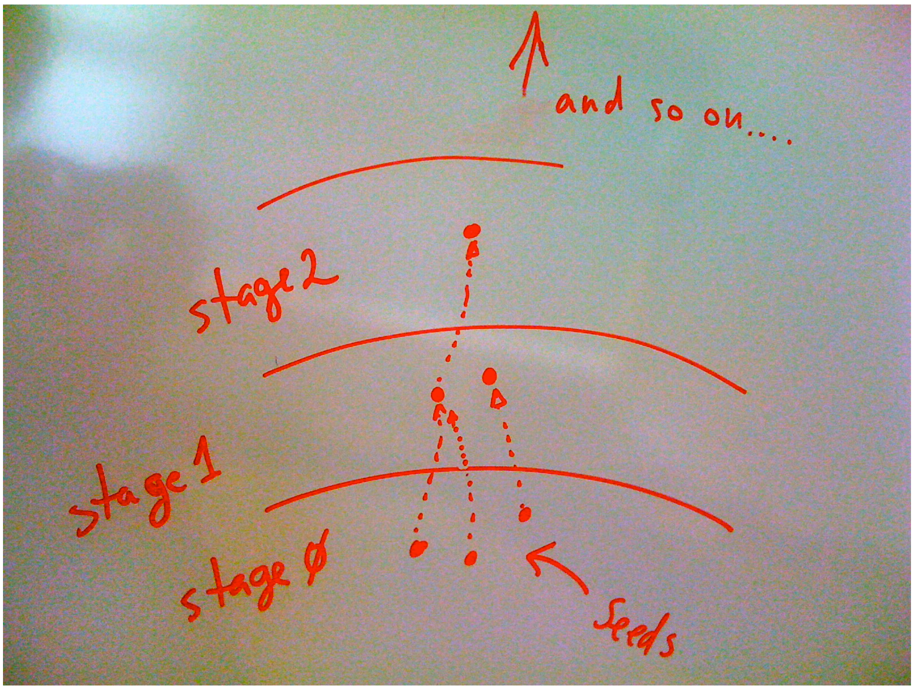

Figure 1: Building a set from the seeds, using rules to generate new elements.

Then, we add a new stage of elements to the set, these "new" elements must be calculated from the "seeds" according to some fixed rule. That will be the second stage.

Then, we add a third stage of elements to the set, calculated from the elements in level two, according to the same fixed rule.

And so on, and on ... indefinitely, to the infinite.

Obviously, we cannot in general implement the whole generation process of the complete inductive set as an algorithm. But the mechanism of induction gives us **for free** the inductive set (which in general contains an infinite number of elements) for granted automatically: we just have to say what are the seeds, and what are the generation rules. Both the seeds and the rules are finite in number, and we can easily write them down.

As a first, example, we consider an inductive definition of the set of natural numbers (supposing that it was not yet defined). First the "seed" (there is only one seed in this case, which is the simplest natural number, namely 0). We thus define:

$$\Longrightarrow 0 \in N$$

This "seed" rule asserts that 0 is in the set N, and defines the first layer of

N, which contains just 0. We then need a "construction" rule, that allows us to add new natural numbers to the set, based on elements already defined in previous layers. The rule looks like:

$$x \in N \Longrightarrow succ(x) \in N$$

We may read this construction rule as: If x is an element of the set N, then $succ(x)$ must also be an element of N. Here we have represented by $succ(x)$ the sucessor of x, e.g., $succ(2) = 3$.

The complete inductive definition of N is then as follows:

$$\begin{array}{ll} ZERO: & \Longrightarrow 0 \in N \\ SUCC: & x \in N \Longrightarrow succ(x) \in N \end{array}$$

It contains two rules, one seed rule and one construction rule.

This inductive definition defines a set N, the set that contains **all** the elements and **only** the elements that may be generated by the rules shown.

Note that we gave names to the rules in the inductive definition, the first rule is called *ZERO* and the second rule is called *SUCC*. To name rules in an inductive definitions, we may invent illustrative names, there is no fixed recipe to give names to rules.

In general, an inductive definition may include any number of seed rules and any number of construction rules, as we will see in forthcoming examples, although in the simple example we have only one seed rule and one construction rule.

A fundamental property of any inductively defined set S is that ANY element $e \in S$ is always justified by a finite number of applications of construction rules, always starting from one or more seed rules.

For example, we have $4 \in N$.

What is the justification of the fact $4 \in N$, according to the inductive definition given above?

It is easy:

- We know that $0 \in N$ by the ZERO (seed) rule!
- We conclude $1 \in N$ by applying the SUCC (construction) rule to $0 \in N$.
- We conclude $2 \in N$ by applying the SUCC (construction) rule to $3 \in N$.
- We conclude $3 \in N$ by applying the SUCC (construction) rule to $2 \in N$.
- We conclude $4 \in N$ by applying the SUCC (construction) rule to $3 \in N$.

We will now go through a sequence of examples of inductive definitions of sets.

Remember that in set theory, a data domain is a set, a function is a set, a relation is a set, and we can also model properties as a set. We will show below how the basic technique of inductive definitions can be used to inductively define functions, relations, properties, data domains, and so on!

1. Example: Even numbers

   Consider the set $EVENN$ of even natural numbers. We have already provided a definition of $EVENN$ using comprehension. We now provide an alternative inductive definition.

   $$\begin{array}{ll} \mathit{ZERO}: & \Longrightarrow 0 \in \mathit{EVENN} \\ \mathit{DUP}: & x \in \mathit{EVENN} \Longrightarrow x+2 \in \mathit{EVENN} \end{array}$$

2. Example: An inductively defined data type

   Consider the set of all finite sequences of natural numbers SEQ. A sequence may be represented in set theory by an n-tuple (see Section 1(16)).

   Let us now define SEQ using an inductive definition (is is not really possible to precisely define this set using either set enumeration or set comprehension / separation).

   $$\begin{array}{ll} \text{EMPTY}: & \Longrightarrow \emptyset \in \mathit{SEQ} \\ \text{ONEMORE}: & s \in \mathit{SEQ}\ ;\ x \in \mathit{NAT} \Longrightarrow (x,s) \in \mathit{SEQ} \end{array}$$

   The (seed) rule EMPTY introduces the empty sequence (represented here by the empty set) in the set SEQ.

   The (construction) rule ONEMORE introduces a new sequence in the set SEQ by adding an arbitrary natural number as the new first element to an already introduced sequence.

   Notice that the ONEMORE rule constructs a new element in the set SEQ not only from some existing element $s \in SEQ$, but also from any existing element $n \in NAT$.

   For example, here is the justification that $(3,4,2,4) \in SEQ$.

   - We know that $() \in SEQ$ by the EMPTY rule!
   - We conclude $(4,\emptyset) \in SEQ$ by applying the ONEMORE rule to $() \in SEQ$ and $4 \in NAT$. Notice that $(4,\emptyset) = (4)$.

- We conclude $(2, (4, \emptyset)) \in SEQ$ by applying the ONEMORE rule to $(4, \emptyset) \in SEQ$ and $2 \in NAT$. Notice that $(2, (4, \emptyset)) = (2, 4)$.
- We conclude $(4, 2, 4) \in SEQ$ by applying the ONEMORE rule to $(2, 4) \in SEQ$ and $4 \in NAT$.
- We conclude $(3, 4, 2, 4) \in SEQ$ by applying the ONEMORE rule to $(4, 2, 4) \in SEQ$ and $2 \in NAT$.

3. General form of Induction Rules

   The last example shows the general format of rules in an inductive definition, which is as follows

   $$e_1 \in S_1 \ ; \ e_2 \in S_2 \ ; \ \cdots e_n \in S_n \Longrightarrow e \in U$$

   Each set S_i is either the name of the set U being inductively defined, or a set expression denoting any **already defined** set.

   The conditions $e_1 \in S_1, e_2 \in S_2, \cdots, e_n \in S_n$ are called the premises of the rule, and the $e \in U$ is called the conclusion of the rule.

4. Example: Inductively defined functions

   As we know a function is a set, a set of ordered pairs subject to the "functional" condition (see Section 1 (19)).

   We may define a function inductively as we have done above for sets.

   Let us illustrate the idea with the Fibonacci function. The Fibonacci function maps $n \in NAT$ to the n^{th} element in the Fibonacci sequence of natural numbers. Remember (this has to do with rabbits ☺) that the Fibonacci sequence is defined as follows. The first and second element in the sequence are both 1. From then on, the n^{th} element in the Fibonacci sequence is computed as the sum of the two previous ones.

   $$1, 1, 2, 3, 5, 8, \cdots$$

   The Fibonnaci function *fib* then gives

   $$\begin{aligned} &fib(0) = 1 \\ &fib(1) = 1 \\ &fib(2) = 2 \\ &fib(3) = 3 \\ &fib(4) = 5 \\ &etc... \end{aligned}$$

To define a function (such as fib) as an inductive set, we need to define a set of ordered pairs $a \mapsto b$ where a is an argument value and b is the corresponding function result.

In the case of the fib function is particularly easy to take care of with an inductive definition, because it is very clear what is the stage by stage construction rules needed!

First we need to introduce the two first values. It is clear that none of these values are computed from the other, they are both seeds, really.

$$\Longrightarrow 0 \mapsto 1 \in fib$$
$$\Longrightarrow 1 \mapsto 1 \in fib$$

Recall that the function fib we are defining is a set of ordered pairs. The two rules above state that the set fib must contain the pairs $0 \mapsto 1$ and $1 \mapsto 1$. This means that $fib(0) = 0$ and $fib(1) = 1$. Actually, we could have written the rules above as

$$\Longrightarrow fib(0) = 1$$
$$\Longrightarrow fib(1) = 1$$

since saying $a \mapsto b \in F$ is the same as saying $F(a) = b$.

Now we need a construction rule, to generate new values for the function fib from "previous" ones. For the Fibonacci function, the rule is simply

$$n \mapsto a \in fib\ ;\ n+1 \mapsto b \in fib \Longrightarrow (n+2) \mapsto (a+b) \in fib$$

This says that if we already know that (at a previous stage) $fib(n) = a$ and $fib(n+1) = b$, then we can define that $fib(n+2) = a+b$.

$$fib(n) = a; fib(n+1) = b \Longrightarrow fib(n+2) = a+b$$

We now summarize our inductive definition for the function fib, now labeling the rules with names.

FIB0 :	$\Longrightarrow fib(0) = 1$
FIB1 :	$\Longrightarrow fib(1) = 1$
FIBNEXT :	$fib(n) = a; fib(n+1) = b \Longrightarrow fib(n+2) = a+b$

We can for example check that $fib(4) = 5$ by writing down the justification, in terms of the available induction rules.

(a) We conclude $0 \mapsto 1 \in fib$ by the FIB0 rule.

(b) We conclude $1 \mapsto 1 \in \mathit{fib}$ by the FIB1 rule.

(c) We conclude $2 \mapsto 2 \in \mathit{fib}$ by applying the FIBNEXT rule to $0 \mapsto 1 \in \mathit{fib}$ and $1 \mapsto 1 \in \mathit{fib}$ introduced in (a) and (b).

(d) We conclude $3 \mapsto 3 \in \mathit{fib}$ by applying the FIBNEXT rule to $1 \mapsto 1 \in \mathit{fib}$ and $2 \mapsto 2 \in \mathit{fib}$ introduced in (b) and (c).

(e) We conclude $4 \mapsto 5 \in \mathit{fib}$ by applying the FIBNEXT rule to $2 \mapsto 2 \in \mathit{fib}$ and $3 \mapsto 3 \in \mathit{fib}$ introduced in (c) and (d).

5. Example: The *sumupto* function

We seek an inductive definition of the *sumupto* function such that

$$\mathit{sumupto}(k) = 1 + 2 + 3 + ... + k$$

for any $k \in NAT$. Notice that this "definition" is not a precise one, and uses "hand waving" notation such as "...", etc.

We can provide a precise inductive definition as follows:

$$\begin{array}{ll} \text{SUM0}: & \Longrightarrow 0 \mapsto 0 \in \mathit{sumupto} \\ \text{SUMNEXT}: & n \mapsto s \in \mathit{sumupto} \Longrightarrow (n+1) \mapsto (n+1+k) \in \mathit{sumupto} \end{array}$$

or, perhaps more readably,

$$\begin{array}{ll} \text{SUM0}: & \Longrightarrow \mathit{sumupto}(0) = 0 \\ \text{SUMNEXT}: & \mathit{sumupto}(n) = s \Longrightarrow \mathit{sumupto}(n+1) = n+1+s \end{array}$$

This inductive definition defines the intended function *sumupto*. For example, we may check the justification that $\mathit{sumpupto}(4) = 10$

(a) We conclude $0 \mapsto 0 \in \mathit{sumupto}$ by the SUM0 rule.

(b) We conclude $1 \mapsto 1 \in \mathit{sumupto}$ by applying the SUMNEXT rule to $0 \mapsto 0 \in \mathit{sumupto}$ introduced in (a).

(c) We conclude $2 \mapsto 3 \in \mathit{sumupto}$ by applying the SUMNEXT rule to $1 \mapsto 1 \in \mathit{sumupto}$ introduced in (b).

(d) We conclude $3 \mapsto 6 \in \mathit{sumupto}$ by applying the SUMNEXT rule to $2 \mapsto 3 \in \mathit{sumupto}$ introduced in (c).

(e) We conclude $4 \mapsto 10 \in \mathit{sumupto}$ by applying the SUMNEXT rule to $3 \mapsto 6 \in \mathit{sumupto}$ introduced in (d).

6. Example: The *len* function

   We seek an inductive definition of the *let* function, that given a sequence of naturals, that is, an element of SEQ as defined above in 2, returns the length of the sequence, for example:

   $$len((1,2,3,4)) = 4$$

   So, we expect $len \in SEQ \to NAT$.

   Here is our inductive definition for the function *len*.

   $$\begin{array}{ll} \text{LENEMPTY :} & \Longrightarrow () \mapsto 0 \in len \\ \text{LENONEMORE :} & s \mapsto l \in len \Longrightarrow (h,s) \mapsto (l+1) \in len \end{array}$$

   or, perhaps more readably,

   $$\begin{array}{ll} \text{LENEMPTY :} & \Longrightarrow len(()) = 0 \\ \text{LENONEMORE :} & len(s) = l \;;\; h \in NAT = l \Longrightarrow len(h,s) = (l+1) \end{array}$$

   This definition inductively defines the intended function *len*. For example, we may check the justification that $len((4,2,4)) = 3$.

   Recall that $(4,2,4) = (4,(2,(4,())))$! Then

   (a) We conclude $() \mapsto 0 \in len$ by the LENEMPTY rule.
   (b) We conclude $(4) \mapsto 1 \in len$ by applying the LENONEMORE rule to $() \mapsto 0 \in len$ introduced in (a) and $4 \in NAT$.
   (c) We conclude $(2,4) \mapsto 2 \in len$ by applying the LENONEMORE rule to $(4) \mapsto 1 \in len$ introduced in (b) and $2 \in NAT$.
   (d) We conclude $(4,2,4) \mapsto 3 \in len$ by applying the LENONEMORE rule to $(2,4) \mapsto 2 \in len$ introduced in (c) and $4 \in NAT$.

7. Top-down and Bottom-up justifications.

   In the previous examples, we have given formal justifications of the fact that an element e belongs to an inductive set I by showing the sequence of rule applications that lead from the seed rules (the most basic elements) to the final construction of e.

   This style of presentation is called a **bottom-up** justification.

   For example, the justification

   (a) We conclude $() \mapsto 0 \in len$ by the LENEMPTY rule.

(b) We conclude $(4) \mapsto 1 \in len$ by applying the LENONEMORE rule to $() \mapsto 0 \in len$ introduced in (a) and $4 \in NAT$.

(c) We conclude $(2,4) \mapsto 2 \in len$ by applying the LENONEMORE rule to $(4) \mapsto 1 \in len$ introduced in (b) and $2 \in NAT$.

(d) We conclude $(4,2,4) \mapsto 3 \in len$ by applying the LENONEMORE rule to $(2,4) \mapsto 2 \in len$ introduced in (c) and $4 \in NAT$.

is bottom-up. It starts by the seed rule $len(()) = 0$ and then proceeds by rule application until the conclusion $len((4,2,4)) = 3$ of the justification is reached.

However, we can also provide justifications the other way around.

We do that by starting from the element that we want to justify, and proceeding backwards down to the seed rules.

For example, the following is the top-down version of the justification above for $len((4,2,4)) = 3$.

(a) We conclude $(4,2,4) \mapsto 3 \in len$ because we can apply LENONEMORE rule to $(2,4) \mapsto 2 \in len$ and $4 \in NAT$.

(b) We conclude $(2,4) \mapsto 2 \in len$ because we can apply LENONEMORE rule to $(4) \mapsto 1 \in len$ and $2 \in NAT$.

(c) We conclude $(4) \mapsto 1 \in len$ because we can apply LENONEMORE rule to $() \mapsto 0 \in len$ and $4 \in NAT$.

(d) We have $() \mapsto 0 \in len$ by the LENEMPTY rule.

8. Example: The *concat* function

We seek an inductive definition of the *concat* function. The concat function accepts as arguments two sequences and gives the sequence obtained by concatenating them. For example,

$$\begin{aligned} &concat((),(1,9)) = (1,9)\\ &concat((3,4),(4,6)) = (3,4,4,6)\\ &concat((1),(2)) = (1,2) \end{aligned}$$

Clearly, we have $concat \in (SEQ \times SEQ) \rightarrow SEQ$.

Here is an inductive definition for the function *concat*.

$$\begin{array}{ll} \text{CEMPTY}: & s \in SEQ \Longrightarrow ((),s) \mapsto s \in concat\\ \text{CSTEP}: & (s,v) \mapsto r \in concat \;;\; h \in NAT \Longrightarrow ((h,s),v) \mapsto (h,r) \in concat \end{array}$$

or, perhaps more readably,

$$\begin{array}{ll} \text{CEMPTY :} & s \in SEQ \Longrightarrow concat((), s) = s \\ \text{CSTEP :} & concat((s, v)) = r \ ; \ h \in NAT \Longrightarrow concat(((h, s), v)) = (h, r) \end{array}$$

Let us see the justification that $concat((4,2),(1,4)) = (4,2,1,4)$. Recall that $(4,2) = (4,(2,\emptyset))$, $(4,2,1,4) = (4,(2,(1,(4,\emptyset))))$, etc !

This time, we write a top-down justification:

(a) We conclude $concat((4,2),(1,4) = (4,2,1,4)$ because we can apply the CSTEP rule to $concat((2),(1,4) = (2,1,4)$ and $4 \in NAT$.

(b) We conclude $concat((2),(1,4) = (2,1,4)$ because we can apply the CSTEP rule to $concat((),(1,4) = (1,4)$ and $2 \in NAT$.

(c) We conclude $concat((),(1,4) = (1,4)$ by the the CEMPTY rule.

1.4 Finite Sets, Infinite Sets, and Computability

The notions of finiteness and infiniteness play a central role in the study of computation. Right from the start, it allows us to separate the notion of function (as we have modeled mathematically with sets in the previous sections), which is an idealized concept, and the notion of algorithm, which is a very concrete computational method or machine.

Think of a given function, such as *concat* function defined above as a set, using induction. The *concat* function defined thus is a set of tuples, a correspondence between argument values and results, that defines the extension of the function. This extension is an infinite set, with elements such as

$$\begin{array}{l} ((1,9,2),(2,3)) \mapsto (1,9,2,2,3) \in concat \\ ((),()) \mapsto () \in concat \\ ((1,1,1,1,1),(2,2,2,2,2)) \mapsto (1,1,1,1,1,2,2,2,2,2) \in concat \\ \ldots \end{array}$$

It is impossible two write down the full table of the concat function, obviously. But we can easily believe that such set exists at least in the world of our imagination and the current body of mathematical knowledge.

A very different thing would be an algorithm implementing the function concat. An algorithm is a mechanical process, that must be physically realizable by some kind of machine, build from a finite set of resources, consuming a finite set of energy for each run, and so on. An algorithm does not contain an infinite lookup table of correspondences between inputs and outputs, it

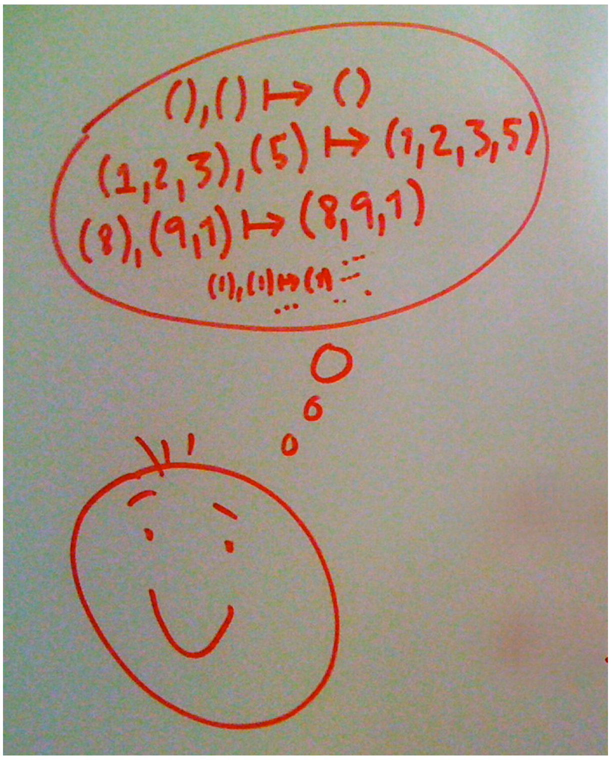

Figure 2: Me thinking about all the concat function pairs $(a, b) \mapsto ab$.

actually needs to calculate the output from the input, using a series of mechanical finite operations, for example, using the instructions of a processor inside a computer.

So an algorithm for computing *concat* must describe a very concrete and physically realizable information manipulating process, defined with some kind of machine or programming language. Such process, given two concrete sequences can laboriously compute a new sequence consisting of their concatenation. It is easy to define such a process, even in a programming language independent way:

1. pick the two input sequences s_1 and s_2

2. count the elements in s_1 and s_2 giving say l_1 and l_2

3. allocate space for a constructing a new sequence r long enough to keep the result $(l_1 + l_2)$ elements.

4. copy the elements of s_1 in sequence to the first l_1 elements of r.

5. copy the elements of s_2 in sequence to the elements of r in positions $l_1, l_1 + 1, \cdots, l_{1+l_2}$.

6. output the sequence r

We have then discussed two very different concepts:

- a function that specifies the concat function, which is a mathematical specification object, consisting of an infinite amount of information, and
- an algorithm that implements the concat function, which is a mechanical procedure that allows a "dummy" machine to compute its output from any given input.

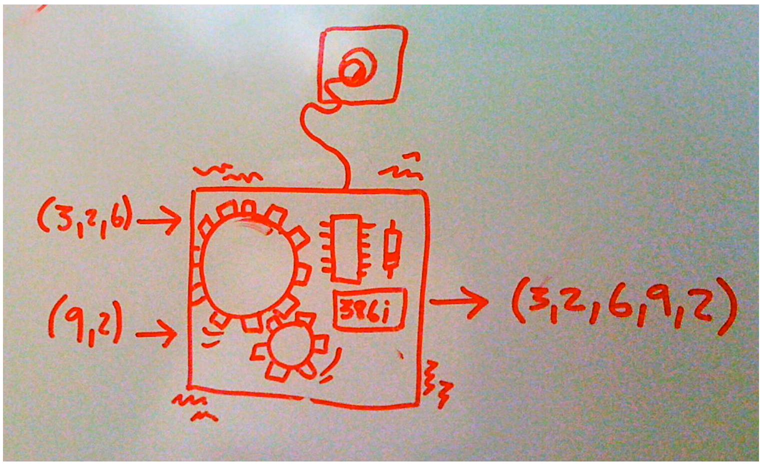

Figure 3: A machine implementing the concat function. It really needs to do some work for each input it is given, there is no place in physical reality for the machine to store an infinite table with all the pairs $(a, b) \mapsto concat(a, b)$.

Another example, may be a function *solution* that given a polynomial with integer coefficients returns *TRUE* if the polynomial has a solution or *FALSE* if it does not. We can imagine that *solution* receives its input in some textual format, and parses it (as say, excel would do).

The function *solution* is very easy to define, say

$$(\exists \vec{u_p} \sum_{i_1}^{p} k_i * u_i{}^{n_i} = v) \Rightarrow (\text{``} \sum_{i_1}^{p} k_i * x_i{}^{n_i} = v\text{''}) \mapsto TRUE \in solution$$

$$(\forall \vec{u_p} \sum_{i_1}^{p} k_i * u_i{}^{n_i} \neq v) \Rightarrow (\text{``} \sum_{i_1}^{p} k_i * x_i{}^{n_i} = v\text{''}) \mapsto FALSE \in solution$$

Every polynomial of the kind considered either has a solution or it does not have a solution, so this is a perfectly well defined total function.

But is there a well defined algorithm implementing this function ?

Indeed, may we find an algorithm actually able to compute the solution function, and effectively check, in a finite number of steps of computation, if any polynomial given as input as a solution or not?

This kind of question may be answered in several ways.

On possibility is to really give a description of an algorithm, as we did above for the concat function, or in some programming language. We may then convince ourselves that the algorithm is correct, and that's all.

Other possibility is that the current state of knowledge does not yet help us to define the algorithm, but it may be the case that some algorithm may be found, if we get smart enough.

Other possibility is that it will never be possible to give an algorithm, just because the solution function cannot be computed by algorithmic means. In this case, we are not thinking of any limitation of the current state of the knowledge, or that we are not smart enough currently to come up with an algorithm, but of the absolute impossibility of solving the problem using any finite method of computation whatsoever!

There are necessarily many functions we can reasonably conceive that cannot be implemented by algorithmic means.

The simplest way to see this is just by counting: there are many more possible functions that algorithms!

Hum .. wait?! How can that be?

Both the set of possible functions and of algorithms is infinite!

How can we say that there are more possible functions than algorithms?

To answer this questions, we need to review a bit the theory of infinite cardinals, and get a better grasp of what does it mean to be "finite" and "infinite". We will also learn that there are several levels of infinity, and that the cardinal of the set of natural numbers is but the "smallest" infinite, in an infinitely increasing sequence of infinite cardinal numbers.

Understanding these basis concepts of infinity is very important to recognize whether a problem is computable by an algorithm, or if it is not (in which case we call the problem "undecidable").

2 Computational Machines and Specifications

Motivation. TBD.

2.1 Deterministic Finite Automata

1. Deterministic Finite Automata (DFA) are very simple models of computation. Each DFA represents a computational system (software or hardware) characterized by a finite set of states, and that evolves from state to state by performing actions (also called transitions), also selected from a finite set of possible transitions.

   Formally, a DFA A is a structure

   $$A = \langle S, \Sigma, s, \delta, F \rangle$$

   where

   (a) S is the finite set of states of A

   (b) Σ is the finite set of actions (or symbols) of A

   (c) s is state in S, the initial state of A

   (d) δ is the transition function of A, that given a current state in S and an action/symbol in Σ, indicates the (unique) next state in S to which the automata should transition.

   So we have $\delta \in S \times \Sigma \to S$ and δ is in general a partial function.

   (e) F is a subset of S, the set F of final states of A.

   For example, the DFA sketched in Figure above may be formally represented, as explained in the previous definition, as the following structure:

   $$\begin{array}{lcl} AFILE & = & \langle S, \Sigma, s, \delta, F \rangle \\ \\ S & = & \{s_1, s_2, s_3\} \\ \Sigma & = & \{\texttt{open}, \texttt{read}, \texttt{write}, \texttt{close}\} \\ s & = & s_1 \\ \delta & = & \{(s_1, \texttt{open}) \to s_2, (s_2, \texttt{read}) \to s_2, \\ & & (s_2, \texttt{write}) \to s_2, (s_2, \texttt{close}) \to s_3\} \\ F & = & \{s_3\} \end{array}$$

It is some times practical to represent the transition function of an DFA by a matrix or array, as we illustrate below:

	open	write	read	close
s_1	s_2	−	−	−
s_2	−	s_2	s_2	s_3
s_3	−	−	−	−

The entries where $\delta(s, a)$ is undefined are marked $-$.

2. A computation of a DFA is any sequence of actions it may perform, starting from the initial state s, and leading to any final state in F.

   For example, the DFA $AFILE$ above has, among many others, the following computations:

   ```
   open close
   open read read close
   open read write read close
   ```

   We may represent the system configuration of a DFA by a pair consisting of a marked sequence of symbols, and the current state. In a marked sequence of symbols we mark with a vertical bar | the separation between the actions / symbols already performed and the actions / symbols still to be performed, and where the first symbol to the right of | is the next to be processed. We may imagine that the current state is the analogous of the program counter, in a standard processor, and the marked sequence of actions some kind of input buffer.

   So, for our current example with the DFA AFILE, we may consider the initial system configuration

   $$(|\texttt{open close}, s_1)$$

   After one transition, because $\delta(s_1, \texttt{open}) = s_2$, the configuration evolves to

   $$(\texttt{open}|\texttt{close}, s_2)$$

   After one more transition, because $\delta(s_2, \texttt{close}) = s_3$, the configuration evolves to

   $$(\texttt{open close}|, s_3)$$

   Since we have reached the end of the sequence and s_3 is a final state ($s_3 \in F$), we can say that the sequence of symbols `open close` is a computation of $AFILE$. There are several different ways to say this, namely

- the sequence `open close` is a computation of $AFILE$.
- the sequence `open close` is accepted by $AFILE$.

We may use any of these ways, depending on the context.

On the other hand, the following sequences of symbols / actions are not computations of $AFILE$.

```
open read
open read open close
```

In the first case, we may try to obtain a computation starting from the configuration

$$(|\texttt{open read}, s_1)$$

After on step, we get to

$$(\texttt{open}|\texttt{read}, s_2)$$

After other step, we get to

$$(\texttt{open read}|, s_2)$$

We have reached the end of the sequence of symbols, but did not reach a final state. Indeed, the computation attempt did not terminate, it got stuck in the non-final state s_2. So, the sequence `open read` is not accepted by the DFA AFILE.

Likewise, consider the sequence `open read open close`. We start off as

$$(|\texttt{open read open close}, s_1)$$

After one step, we get to

$$(\texttt{open}|\texttt{read open close}, s_2)$$

After one more step, we get to

$$(\texttt{open read}|\texttt{open close}, s_2)$$

Now the problem is that the next symbol to process in `open`, but unfortunately the transition function δ does not define any valid transition. Indeed $\delta(s_2, \texttt{open})$ is not defined! So, this computation atempt did not properly terminate as well, and we conclude that the sequence `open read open close` is not accepted by the DFA $AFILE$.

3. DFAs are very convenient also because they lead to very efficient and simple implementations of trace recognizers. Given an array representation of the transition function, we may simply implement the Java like pseudo-code below, to check if an input sequence *input* is accepted by the given DFA.

```
boolean accept(Word input) {
  State currentState = initialState;
  while input.hasNext() {
     State nextState = delta[ currentState, input.next() ];
     if (nextState.undefined()) return false;
     currentState = nextState;
   }
  return (currentState.isFinal())
  }
```

   We have here represented a word by an iterator of symbols. Of course, other representations are possible.

4. For any DFA $A = \langle S, \Sigma, s, \delta, F \rangle$, it makes sense to talk about the set of all computations of Am, or, equivalently, about the set of all sequences of symbols accepted by A. This set of sequences, is called the language accepted by the DFA A, noted $\mathcal{L}(A)$.

$$\mathcal{L}(A) = \{w \in \mathit{Words}(\Sigma) \mid w \text{ is accepted by the DFA } A\}$$

   This is not a precise definition, since we did not yet explain what is the set $Words(\Sigma)$ and what does it really mean to "be accepted". For that, we need to introduce some few concepts.

2.2 Alphabets, words, traces, and (formal) languages

We have talked about sets of actions, sequences of actions, symbols, etc, when discussing DFAs.

All these notions involving sequences of symbols, are abstracted and studied in theoretical computer science by the notion of "formal language". The concept of formal language is very general and useful, and is not just connected to "languages" in the usual sense. We will give some examples of applications belows. But before, we introduce a few useful definitions.

1. Alphabet

An alphabet is just a finite set of symbols (also actions). We may define alphabets as we wish, typically by enumeration. For example,

$$\begin{array}{lcl} \Sigma_{FILE} & = & \{\texttt{open}, \texttt{read}, \texttt{write}, \texttt{close}\} \\ DIGITS & = & \{\texttt{0}, \texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}, \texttt{5}, \texttt{6}, \texttt{7}, \texttt{8}, \texttt{9}, \} \end{array}$$

2. Word (or trace)

   A word (or trace) over an alphabet Σ is a finite, possibly empty, sequence of symbols taken from Σ.

   We write () for the empty word. Some texts also represent the empty word by λ or ϵ.

   Here are some examples:

   `close close write write` is a word over Σ_{FILE}.

   `open read close close` is a word over Σ_{FILE}.

   `open 1 2 close` is not a word over Σ_{FILE}.

   () is a a word over Σ_{FILE}.

   `1 1 1 1` is a word over $DIGITS$.

   () is a word over $DIGITS$.

   `1 a b 1` is not a word over $DIGITS$.

3. Extending one word by one symbol.

   Given a word u over Σ and a symbol $a \in \Sigma$, we denote by au the word over Σ obtained by adding a before the first symbol of u.

   For example, if u is `1 1 1 1` and a is `3` then au is `3 1 1 1 1`.

   For example, if u is () and a is `2` then au is `2`.

   Note that in rigor one thing is a symbol, and other is a word with just one symbol (like one thing is an integer, and other thing is a list with length one containing the same integer). We shall never make confusions, though.

4. The set $Words(\Sigma)$ of all words (or traces) over the alphabet Σ.

   Given an alphabet Σ, we may easily define the set $Words(\Sigma)$ of all words over the alphabet Σ, using induction, as discussed in the first chapter.

$$\begin{array}{l} \Rightarrow () \in Words(\Sigma) \\ w \in Words(\Sigma) \wedge a \in \Sigma \Rightarrow aw \in Words(\Sigma) \end{array}$$

It should be pretty clear that whenever the alphabet Σ is non empty, the set $Words(\Sigma)$ of all words over Σ is countably infinite.

We may now define what is a (formal) language over an alphabet.

5. Formal language over an alphabet Σ

   Given an alphabet Σ, a formal language over Σ is any subset L of $Words(\Sigma)$. So the set $Lang(\Sigma)$ of all languages over Σ is

$$Lang(\Sigma) = \wp(Words(\Sigma))$$

   For a simple example, we may consider the set *PRIMES* of all words in $Words(DIGITS)$ that represent prime numbers. This is the example of what we cal a "formal language". Then, we know that

$$\texttt{11} \in PRIMES$$

   but

$$\texttt{22} \notin PRIMES$$

   although $\texttt{11} \in DIGITS$ and $\texttt{22} \in DIGITS$.

   As another example, we may consider the set $\mathcal{L}(AFILE)$ of all words over Σ_{FILE} that are accepted by the DFA FILE, presented above. Then, we know that

$$\texttt{open read close} \in \mathcal{L}(AFILE)$$

   but

$$\texttt{open open} \notin \mathcal{L}(AFILE)$$

   even if clearly $\texttt{open read close} \in \Sigma_{FILE}$ and $\texttt{open open} \in \Sigma_{AFILE}$.

   These are two simple examples of formal languages. Obviously, a language may be finite or infinite. It is easy to see that both *PRIMES* and $\mathcal{L}(AFILE)$ are infinite languages.

   We may now get back to AFDs, and to the precise definition of what does it mean for a word / trace to be accepted by an automaton.

2.3 Language Accepted by a DFA

6. Acceptance of a word by a AFD A

   Given any DFA $A = \langle S, \Sigma, s, \delta, F \rangle$, we may define a relation

$$step_A \subseteq S_A \times Words(\Sigma_A) \times S_A$$

that represents the multistep transition relation for the DFA A.

Intuitively, we want the relation $step_A$ to be defined so that $(s, w, s') \in step_A$ if and only if the DFA A can transition from state s to state s' by processing the symbols in w in the indicated sequence.

For example, getting back to our running example DFA $AFILE$, we would have

$$
\begin{array}{l}
(s_2, (), s_2) \in step_{AFILE} \\
(s_1, \texttt{open close}, s_3) \in step_{AFILE} \\
(s_1, \texttt{open close}, s_2) \notin step_{AFILE} \\
(s_2, \texttt{read read}, s_2) \in step_{AFILE} \\
(s_2, \texttt{read read write}, s_2) \in step_{AFILE} \\
(s_2, \texttt{open}, s_2) \notin step_{AFILE}
\end{array}
$$

Given any DFA A, it is easy to define the relation $step_A$ inductively as follows

$$
\begin{array}{l}
s \in S_A \Rightarrow (s, (), s) \in step_A \\
(s', u, s'') \in step_A \wedge a \in \Sigma_A \wedge \delta_A(s'', a) = s' \Rightarrow (s, au, s'') \in step_A
\end{array}
$$

7. Language accepted by an AFD A. Given the concepts presented above we may now give a precise definition for the language $\mathcal{L}(A)$ accepted by an AFD $A = \langle S, \Sigma, s, \delta, F \rangle$, intuitively described above by

$$\mathcal{L}(A) = \{w \in Words(\Sigma) \mid w \text{ is accepted by the DFA } A\}$$

Indeed, we may simply set

$$\mathcal{L}(A) = \{w \in Words(\Sigma) \mid \exists f. f \in F_A \wedge (s_A, w, f) \in step_A\}$$

Reading off this precise definition, $\mathcal{L}(A)$ is the set of words that can induce a sequence of computation steps from the initial state s_A to some final state $f \in F_A$, as conveniently expressed by the $step_A$ relation.

2.4 Regular Languages

A language for which it is possible to construct a DFA that accepts it is called a **regular language**.

DFAs have limited expressive power, mainly because they only possess a finite number of states. There are many languages which can not be accepted by DFAs. For example, the language *PRIMES* mentioned above is not regular.

Other simple example of non-regular languages are the so-called "parentesis" languages. A "parentesis" language is a language with a nested block structure, like the language with arithmetic expressions with parentheses $(1 + ..(...)... * 3)$, or a programming language with nested blocks, such as Java $\{\{...\}\}$. Intuitively, to make sure that all opening parenthesis are matched by corresponding closing parentesis, a DFA would need to be prepared to "know" in every state how many parenthesis are currently open. But since a DFA only contains a finite number of states N, it could never handle an expression with say $N + 10$ nested blocks of parenthesis. We will study later in this course more powerful (stack-based) machines, which will be able to accept non-regular languages as the ones illustrated here.

Still, there many interesting computational systems that may be modeled by DFAs, and the simplicity of DFAs are attractive and convenient.

For example, the language of all words over $DIGITS$ that represent numbers divisible by k, where k is any natural number, is regular.

The language over the UNICODE character set that represents all valid descriptions of floating numbers in IEEE754 format is regular.

You will see many other examples in the exercises.

2.5 Regular Expressions

2.6 Compiling any Regular Expression to a DFA

2.7 Non Deterministic Finite Automata

2.8 Compiling any Regular Expression to a NFA

2.9 Compiling a NFA to an equivalent DFA