Bengt Nordström,
Department of Computing Science,
Chalmers and University of Göteborg,
Göteborg, Sweden

November 4, 2013

1 Some mathematical preliminaries: sets, relations and functions

1.1 Propositions, predicates and relations

A mathematical *propositon* – also known as a formula – is something we prove. A proposition is true when we have a proof of it[1] , and is false when its negation is true. Notice that there are mathematical propositions which we do not know whether they are true or false (for instance Goldbach's conjecture, or if there are 52 consecutive 3s in the decimal expansion of π.). A proposition is built up by logical connectives like $\forall, \exists, =, \supset, \wedge, \vee \ldots$, but it is also possible to define new ones by using an inductive definition. We will see examples of this later.

A *predicate* P over a set A is a propositional function over the set. This means that for each element x in A we get a proposition $P(x)$. A simple example is the predicate $\mathtt{iszero}$, which is a propositional function over N expressing that its argument is equal to 0. So $\mathtt{iszero}(n)$ is a proposition for each $n \in \mathrm{N}$. We see that a predicate expresses a property of the elements in the set.

A *relation* is a generalization of a predicate, instead of one argument it can take several arguments. Simple examples are the equality relation over a set and ordering relations over sets. An example from this course is the relation $p \longrightarrow q$, which expresses that the program p computes to q in one or more steps.

We will use inductively defined set and relations. This seem very natural for computer scientists and it avoids a lot of unnecessary coding. The tradition in mathematics (recursive functions, lambda calculus, Turing machines) is to repre-

[1]The reader who is interested in the foundation of mathematics will notice that this is the constructive definition of truth.

sent an element in an inductively defined set as a number. This is a very indirect representation.

1.2 Inductive sets

To give an inductive definition of a set amounts to give an exhaustive list of rules how to form elements in the set. There is a unique constructor associated with each clause. This is a completely natural for a computer scientist used to functional languages like Haskell, Caml and SML.

The canonical example of an inductively defined set is the set N of natural numbers.

1.2.1 Natural numbers

Definition 1 (N) *The set* N *of natural numbers is inductively defined by the following clauses:*

$$0 \in \mathrm{N}$$

$$s(x) \in \mathrm{N} \text{ , } \textit{if } x \in \mathrm{N}$$

This is to be read as:

0 is a natural number and if x is a natural number, then $s(x)$ is a natural number. The constructors are 0 and s. Any natural number can be obtained by applying one of these two laws a finite number of times. In some presentations there is an extra clause expressing exactly this requirement, we will instead consider it to be part of the meaning of an inductive definition that the clauses are exhaustive and applied a finite number of times [2] .

An element in an inductively defined set has always (in our presentation) the form $c(e_1, \ldots e_n)$ for $n \geq 0$. We also assume that two elements $c(e_1, \ldots e_n)$ and $c'(e'_1, \ldots e'_m)$ are equal if and only if the constant c is the same as the constant c', $n = m$, and $e_i = e'_i$ for all $i \leq n$. This condition of equalities simplifies the

[2] Inductively defined types in the languages above do not have the restriction of finite generation. This is unfortunate, since it destroys many properties which we are used to. For instance that $x \neq s(x)$.

presentation (for instance simplifying the recursion scheme) but it also makes the definition of some sets a little less straightforward. It is for instance erroneous to define the set of integers by the clauses:

$$0 \in Z$$
$$s(x) \in Z \text{ , if } x \in Z$$
$$p(x) \in Z \text{ , if } x \in Z$$

with the intention that s and p are the successor and predecessor operations [3].

We will often be liberal about the syntax for the constructors and use prefix, postfix or mixfix syntax.

Induction over N

From the inductive definition of N it is clear that we can use the following proof principle to prove that a predicate C over the natural numbers is true for any natural number p:

To prove $(\forall p \in N)C(p)$ it is enough to prove

- $C(0)$
- $(\forall x \in N)C(x) \supset C(s(x))$

The sign $\supset$ means implication. We see that we get one proof condition for each clause in the inductive definition of N. We have reduced the problem of proving a property about any natural number to two cases, one of the zero case and one for the successor case.

Structural recursion over N

From the inductive defintion it is also clear that it gives us a way to define total functions from N to a set C. If we have the following

- $d \in C$

[3] If you don't see why, look at the previous paragraph again!

- $e \in N \to C \to C$

then we can construct a function $f \in N \to C$ by the following definition

$$f(0) = d$$
$$f(x+1) = e(x, f(x))$$

This scheme for defining functions is called *primitive recursion*, and we can capture it by a higher order function $rec \in N \to C \to (N \to C \to C) \to C$ which will be defined in the following way:

$$rec(0, d, e) = d$$
$$rec(x+1, d, e) = e(x, rec(x, d, e))$$

Now we can use this function to give a non-recursive definition of a primitive recursive function. If we for instance want to define the function f above then we can use the definition

$$f(x) =_{def} rec(x, d, e)$$

Notice that there is no occurrence of f in the right hand side. We can prove by induction that f satisfies the equations. For the base case $f(0) = rec(0, d, e) = d$ by the definition of f and rec. For the induction step we assume that $f(x) = rec(x, d, e)$ and we want to prove that $f(x+1) = e(x, f(x))$. We see that $f(x+1) = rec(x+1, d, e) = e(x, rec(x, d, e) = e(x, f(x))$.

1.2.2 The cartesian product between two sets

Definition 2 (The cartesian product between two sets) *The cartesian product* $A \times B$ *between the two sets* A *and* B *is defined by the following clause:*

$$pair(a, b) \in A \times B \text{ , } \textit{if } a \in A, b \in B$$

So, an element in $A \times B$ is always equal to something on the form $pair(a, b)$, where $a \in A, b \in B$.

Proof principle for $A \times B$

From the definition of the cartesian product it is clear that we can use the following proof principle to prove that a predicate C over $A \times B$ is true for any element $p \in A \times B$:

To prove $(\forall p \in A \times B)C(p)$ it is enough to prove

- $(\forall x \in A, y \in B)C(\text{pair}(x, y))$

So, we have reduced the problem of proving a property of an element in $A \times B$ to proving a property about elements in A and B.

Structural recursion over $A \times B$

From the definition it is also clear that it gives us a way to define functions from $A \times B$ to a set C. If we have the following

- $e \in A \to B \to C$

then we can construct a function $f \in A \times B \to C$ by the following definition

$$f(\text{pair}(x, y)) = e(x, y)$$

So, we have reduced the problem of defining a function from $A \times B$ to the problem of defining a function from A and B.

This scheme for defining functions can be captured by a higher order function $\text{split} \in A \times B \to (A \to B \to C) \to C$ which will be defined in the following way:

$$\text{split}(\text{pair}(x, y)e) = d$$

Now we can use this function to give a definition of f above without using pattern matching:

$$f(x) =_{\text{def}} \text{split}(x, e)$$

1.2.3 Lists

Definition 3 (Lists) *The set* $\mathrm{List}(A)$ *of lists of elements from the set* A *is inductively defined by the following clauses:*

$$\mathrm{nil} \in \mathrm{List}(A)$$
$$\mathrm{cons}(a, as) \in \mathrm{List}(A) \text{ , } \textit{if } a \in A, as \in \mathrm{List}(A)$$

So, a list is either equal to nil or to $\mathrm{cons}(a, as)$, where $a \in A$ and $as \in \mathrm{List}(A)$. Notice that the inductive definition means that all lists are finite.

Proof principle for $\mathrm{List}(A)$

From the definition of $\mathrm{List}(A)$ it is clear that we can use the following proof principle to prove that a predicate C over $\mathrm{List}(A)$ is true for any list p:

To prove $(\forall p \in \mathrm{List}(A))C(p)$ it is enough to prove

- $C(\mathrm{nil})$
- $(\forall a \in A, as \in \mathrm{List}(A))C(as) \supset C(\mathrm{cons}(a, as))$

We have one case for each way of forming elements in $\mathrm{List}(A)$.

Structural recursion over $\mathrm{List}(A)$

From the definition it is also clear that it gives us a way to define functions from $\mathrm{List}(A)$ to a set C. If we have the following

- $d \in C$
- $e \in A \to \mathrm{List}(A) \to C \to C$

then we can construct a function $f \in \mathrm{List}(A) \to C$ by the following definition

$$f(\mathrm{nil}) = d$$
$$f(\mathrm{cons}(a, as)) = e(a, as, f(as))$$

This scheme for defining functions is called *primitive recursion over lists*, and we can capture it by a higher order function

$$\text{listrec} \in \text{List}(A) \to C \to (A \to \text{List}(A) \to C \to C) \to C$$

which will be defined in the following way:

$$\begin{aligned} \text{listrec}(\text{nil}, d, e) &= d \\ \text{listrec}(\text{cons}(a, as), d, e) &= e(a, as, \text{listrec}(as, d, e)) \end{aligned}$$

So, the function f above can be defined in a non-recursive way by

$$f(x) =_{\text{def}} \text{listrec}(x, d, e)$$

1.2.4 Bool

The set of boolean values can be defined in a similar way. We leave this as an exercise.

1.2.5 Tables

If A and B are sets, then we make the following inductive definition of the set $\mathbf{T}(A, B)$ of tables from A to B:

We assume that $a \in A$, $b \in B$ and $t \in \mathbf{T}(A, B)$.

$$\begin{aligned} \text{nilt} &\in \mathbf{T}(A, B) \quad \text{the empty table} \\ \text{upd}(t, a, b) &\in \mathbf{T}(A, B) \quad \text{a non-empty table} \end{aligned}$$

We leave it as an exercise to formulate the proof principle for tables.

1.3 Inductive relations

In the same way as we define sets inductively, we will define relations inductively. Remember that we look at a relation as a propositional function, i.e. something which when given its argument(s) yields a proposition.

As an example, the relation which expresses the relation that the result of looking up the key a in the table t is b can be defined by the two clauses:

$$\text{lookup}(\text{upd}(t, a, b), a, b)$$

$$\frac{\text{lookup}(t, a, b) \qquad a \neq a'}{\text{lookup}(\text{upd}(t, a', b'), a, b)}$$

Here we assume that $t \in \mathbf{T}(A, B)$, $a \in A$, and $b \in B$.

We read this as an inductive definition, i.e. that the only way we can get a direct proof that $\text{lookup}(t, a, b)$ holds is by using one of these clauses.

1.4 Partial functions

We say that f is a partial function from the set A to the set B, $f \in A \tilde{\to} B$, if f is a binary relation over A and B, such that $f(a, b)$ and $f(a, c)$ implies $b = c$. We write $f(a) = b$ instead of $f(a, b)$. We will say that $f(a)$ is defined if there is an element b such that $f(a, b)$. Notice that the lookup-relation is a partial function in its first two arguments.

A function $f \in A \tilde{\to} B$ is *total* if it is defined for all its arguments:

$$\forall x \in A \,.\, \exists b \in B \,.\, f(a) = b$$

The function is *surjective* if all elements in B are mapped:

$$\forall b \in B \,.\, \exists a \in A \; f(a) = b$$

The function is *injective* if it always maps different objects in A to different objects in B:

$$a \neq a' \supset f(a) \neq f(a')$$

or equivalently

$$f(a) = f(a') \supset a = a'$$