Math Preliminaries

Introduction

Ryan Stansifer

Department of Computer Sciences
Florida Institute of Technology
Melbourne, Florida USA 32901

http://www.cs.fit.edu/~ryan/

19 January 2017

De Morgan Laws

$$P \text{ and } Q \text{ iff } \text{not}((\text{not } P) \text{ or } (\text{not } Q))$$

$$P \text{ or } Q \text{ iff } \text{not}((\text{not } P) \text{ and } (\text{not } Q))$$

$$A \cup B = \overline{\overline{A} \cap \overline{B}}$$

$$A \cap B = \overline{\overline{A} \cup \overline{B}}$$

Leopold Kronecker (1823–1891)

Jasper Johns
Numbers in Color

Leopold Kronecker (December 7, 1823 – December 29, 1891) was a German mathematician and logician who argued that arithmetic and analysis must be founded on “whole numbers,” and was quoted by Heinrich Weber as having said, “God made the integers; all else is the work of man.” Kronecker’s finitism made him a forerunner of intuitionism in foundations of mathematics.

Eine wesentliche Lücke würde aber in dem Bilde des Mathematikers Kronecker bleiben, wenn ich seine Stellung zu den fundamentalen, philosophischen Fragen der Mathematik mit Stillschweigen übergehen wollte. Es ist ein Standpunkt, der besonders in seinen späteren Jahren hervortrat, vielleicht mehr noch im persönlichen Verkehr als in der Oeffentlichkeit; aber auch öffentlich hat er seine Anschauungen nicht verleugnet und z. B. in der Festschrift zu Zeller's Jubiläum scharf hervorgekehrt.

In Bezug auf Strenge der Begriffe stellt er die höchsten Anforderungen und sucht alles, was Bürgerrecht in der Mathematik haben soll, in die krystallklare eckige Form der Zahlentheorie zu zwängen. Manche von Ihnen werden sich des Ausspruchs erinnern, den er in einem Vortrag bei der Berliner Naturforscher-Versammlung im Jahre 1886 that: „Die ganzen Zahlen hat der liebe Gott gemacht, alles andere ist Menschenwerk".

So war ihm alles, was sich nicht seines arithmetischen Ursprungs unmittelbar bewusst war, unsympathisch, und sein Streben ging dahin, nicht nur in der Algebra, sondern auch in der Functionentheorie die arithmetische Abstammung deutlich hervortreten zu lassen.

Heinrich Martin Weber, "Leopold Kronecker," *Jahresbericht der Deutschen Mathematiker-Vereinigung, Zweiter Band, 1891–92*, Berlin: Druck und Verlag von Georg Reimer, 1893, pages 2–29. Also in *Mathematische Annalen*, volume 43, number 1, 1892, pages 1–25.

Theorem

(Mathematical Induction on Natural Numbers.) Let $P(n)$ be a property that pertains to natural numbers $n \in \mathbb{N}$. If the following are true:

$$P(0) \text{ is true}$$
$$\text{For any } k \in \mathbb{N}, P(k) \text{ implies } P(k+1)$$

Then for any $n \in \mathbb{N}$, $P(n)$ is true.

Example

Theorem

For all $n \in \mathbb{N}$, $\Sigma_{i=1}^{n} = \frac{n(n+1)}{2}$.

Theorem

(Mathematical Induction on Natural Numbers – Noetherian [strong, complete].) Let $P(n)$ be a property that pertains to natural numbers $n \in \mathbb{N}$. If the following are true:

$$P(0) \text{ is true}$$

$$\text{For any } i, k \in \mathbb{N}, \text{ if } i < k \text{ and } P(i), \text{ then } P(k)$$

Then for any $n \in \mathbb{N}$, $P(n)$ is true.

Example

Theorem

For all natural numbers p with $p > 1$, p is the product of one or more prime numbers.

Example

Induction works on natural numbers because the set of natural numbers is well-founded by the less-than relations.

A binary relation $\prec$ on a set X is said to be well-founded or notherian, if every non-empty subset of X has a minimal element with respect to the $\prec$ relation.

$$\text{for all } S \subseteq X,\ S \neq \emptyset \text{ implies for some } m \in S, \text{ and all } s \in S, \text{ not } s \prec m$$

Inductive Set

But it easy to construct sets which are well-founded, these sets are said to be inductive sets and the principle of induction applies to such sets. The natural numbers are a special case.

An inductive definition of a set consists of two steps:

- Basis. Some set elements are explicitly asserted to belong to the set.
- Construction. One or more rules for constructing new elements of the set from existing elements

It is implied that nothing else is in the set except those which can be demonstrated are in the set by the means.
See, Hein, 2nd, Section 3.1 [2]

Example

The natural numbers is the set constructed from zero and the successor constructor.

A string of an alphabet is the set constructed from the empty string and the family of constructors (one for each letter of the alphabet) which extend the string by a letter.

Sometimes inductive sets are described by other means like judgments of Post system proofs, or sentential forms of CFG derivations. Constructing sets these ways leads immediately to using induction to prove properties of all Post system judgments or CFG sentential forms.

Also, the Haskell programming language provides a great way to construct inductive sets by ways of algebraic data types.

Unit

In Haskell new data types can be created by enumerating all the values in them.

```
data Bool = True | False
```

The data type `Bool` has two values: `True` and `False`.
A type may only have one value in it as the Haskell data type called unit:

```
data () = ()
```

The data type and the value of the data have the same notation in Haskell (Since data types are values are not easily confused, this is not a big problem.)

Consider the constructive definition of data values as in this Haskell example:

```
data Nat = Zero | Succ Nat
```

This definition defines two methods for constructing data values of type `Nat`—one of which is simple enumeration and the other is recursive. Constructable values include `Zero`, `Succ(Zero)`, `Succ(Succ(Zero))`, and so on.

Note: These are the only values you can *construct* of type `Nat` in Haskell. But we may have to contend with other expressions of type `Nat` in Haskell. For example, the expression `f(-2)` where

```
f (0) = Zero
f (n) = Succ (f (n-1))
```

has type `Nat`in Haskell *but no value*–it does not terminate.

The same methods of construction can be restated as a Post system:

$$\frac{}{\texttt{Zero} \in \texttt{Nat}} \qquad \frac{v \in \texttt{Nat}}{\texttt{Succ(}v\texttt{)} \in \texttt{Nat}}$$

$$\frac{}{0 \in \mathbb{N}} \qquad \frac{n \in \mathbb{N}}{n' \in \mathbb{N}}$$

Alternatively, we can use what might be called tally notation:

$$\frac{}{N} \qquad \frac{Nx}{Nx\,|}$$

In this system an alternative set of constructable values is created in one-to-one correspondence to the previous values: N, $N\,|$, $N\,||$, $N\,|||$, and so on.

Numerals

These four well-defined systems for constructing numerals are all equivalent representations of the natural numbers.

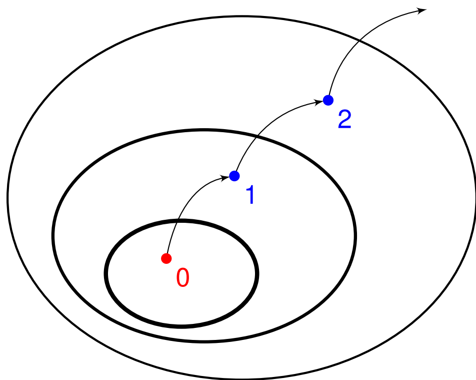
0	`Zero`	0	N
1	`Succ(Zero)`	$0'$	$N\,\vert$
2	`Succ(Succ(Zero))`	$0''$	$N\,\vert\vert$
3	`Succ(Succ(Succ(Zero)))`	$0'''$	$N\,\vert\vert\vert$
4	`Succ(Succ(Succ(Succ(Zero))))`	$0''''$	$N\,\vert\vert\vert\vert$
5	`Succ(Succ(Succ(Succ(Succ(Zero)))))`	$0'''''$	$N\,\vert\vert\vert\vert\vert$

Additionally, yet another formal, description of those values is possible using context-free grammars:

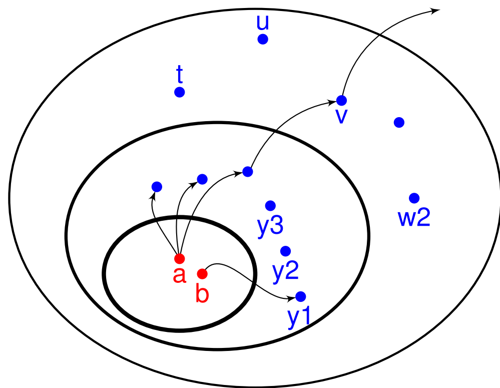
```
1  S → N
2  N → N |
```

The set of sentential forms is exactly the same tally notation as the earlier Post system.

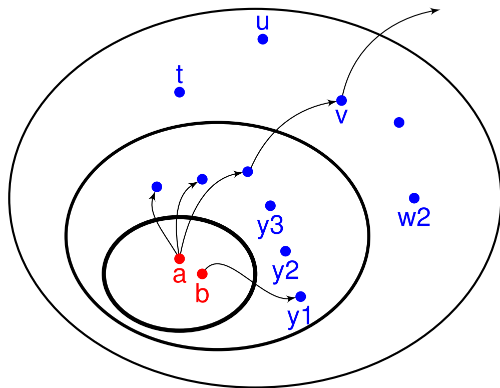
Inductively Defined Structures



Inductively Defined Structures



Inductively Defined Structures (Regular Expressions



Back to proofs by induction

Consider the Haskell data type `Nat`:

```
data Nat = Zero | Succ Nat
```

Theorem

(Mathematical Induction on the Haskell type `Nat`.) Let $P(v)$ be a property that pertains to value v in data type `Nat`. If the following are true:

$$P(\texttt{Zero}) \text{ is true}$$

$$\text{For any } v \in \texttt{Nat},\ P(v) \text{ implies } P(\texttt{Succ}(v))$$

Then for any $v \in$ `Nat`, $P(v)$ is true.

Everything in the inductive type/set is constructed by a finite number of constructor applications.

The strict subterm relation, i.e., the thing/expression/term t constructed from x by any of the constructors is a partial-order. There are some null-ary constructors, i.e., somethings/expressions/terms constructed out of nothing, then the partial-order is a well-founded order, and the induction principle applies.

Lists

In Haskell types have kind `*`. Type constructors (as opposed to value constructors or just constructors) which construct a new kind of type from an existing type have kind `*->*`.
Consider the Haskell data type constructor `Fwd` of kind `*->*`:

```
data Fwd a = F0 | a :> (Fwd a)
```

Consider the Haskell data type `Bwd` of kind `*->*`:

```
data Bwd a = B0 | (Bwd a) :< a
```

Consider the predefined, Haskell data type constructor for polymorphic lists (kind `*->*`):

```
data [a] = [] | a : [a]
```

Consider the Haskell data type $\texttt{Fwd}\,A$:

```
data Fwd a = F0 | a :> (Fwd a)
```

Theorem

(Mathematical Induction on the Type $\texttt{Fwd}\,A$.) Let $P(v)$ be a property that pertains to value v in data type $\texttt{Fwd}\,A$. If the following are true:

> *$P(\texttt{F0})$ is true*
>
> *For all $v \in \texttt{Fwd}\,A$ and for all $a \in A$, $P(v)$ implies $P(a\texttt{:>}v)$*

Then for any $v \in \texttt{Fwd}\,A$, $P(v)$ is true.

Consider the Haskell data type $\texttt{Bwd}\,A$:

```
data Bwd a = B0 | (Bwd a) :< a
```

Theorem

(Mathematical Induction on the Type $\texttt{Bwd}\,A$.) Let $P(v)$ be a property that pertains to value v in data type $\texttt{Bwd}\,A$. If the following are true:

> *$P(\mathit{F0})$ is true*
>
> *For all $v \in \texttt{Bwd}\,A$ and for all $a \in A$, $P(v)$ implies $P(v\,\texttt{:<}\,a)$*

Then for any $v \in \texttt{Bwd}\,A$, $P(v)$ is true.

Consider the predefined, Haskell data type for polymorphic lists (kind `*->*`):

```
data [a] = [] | a : [a]
```

Theorem

(Mathematical Induction on the Haskell lists.) Let $P(v)$ be a property for a list value v. If the following are true:

$$P(\texttt{[]}) \text{ is true}$$
$$\text{For all } v \in [A] \text{ and for all } a \in A,\ P(v) \text{ implies } P(x : v)$$

Then for any $v \in [A]$, $P(v)$ is true.

0	`Zero`	0	N
1	`Succ(Zero)`	$0'$	$N\vert$
2	`Succ(Succ(Zero))`	$0''$	$N\vert\vert$
3	`Succ(Succ(Succ(Zero)))`	$0'''$	$N\vert\vert\vert$
4	`Succ(Succ(Succ(Succ(Zero))))`	$0''''$	$N\vert\vert\vert\vert$

The type `Fwd` and `Bwd` are identical to the pre-defined, polymorphic Haskell list type (of kind `*->*`). In the special case of the pre-defined Haskell unit type:

```
data () = ()
```

lists are isomorphic to natural numbers.

0	`F0`	`B0`	`[]`
1	`():>F0`	`B0:<()`	`():[]`
2	`():>():>F0`	`B0:<():<()`	`():():[]`
3	`():>():>():>F0`	`B0:<():<():<()`	`():():():[]`
4	`():>():>():>():>F0`	`B0:<():<():<():<()`	`():():():():[]`

Consider the Haskell data type $\texttt{Tree}\,A$:

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

Theorem

(Mathematical Induction on the Type $\texttt{Tree}\,A$*.) Let* $P(r)$ *be a property that pertains to value r in data type* $\texttt{Tree}\,A$*. If the following are true:*

- $P(\texttt{Leaf})$ *is true*
- *For all* $v, w \in \texttt{Tree}\,A$ *and for all* $a \in A$*,* $P(v)$ *implies* $P(\texttt{Node}\,v\,a\,w)$

Then for any $r \in \texttt{Tree}\,A$*,* $P(r)$ *is true.*

Consider the Haskell data type `Regex` *A*:

```
data Regex a = Empty | Sym a |
  Alt (Regex a) (Regex a) | Star (Regex a)
```

Some example values of type `Regex` *A* are:

```
Empty
Sym 'a'
Sym 'b'
Alt Empty (Sym 'c')
Star (Alt (Sym 'd') (Sym 'e'))
```

Note: We omitted the final, recursive case in the definition of regular expressions, because it did not add different structure to this example.

The same methods of construction can be restated as a Post system:

$$\frac{}{\emptyset \in Rx} \quad \frac{a \in \Sigma}{a \in Rx} \quad \frac{r_1 \in Rx \quad r_1 \in Rx}{(r_1 + r_2) \in Rx} \quad \frac{r_1 \in Rx}{(r_1)^* \in Rx}$$

Or for that matter a CFG:

1. $R \rightarrow \emptyset$
2. $R \rightarrow a$
3. $R \rightarrow (R + R)$
4. $R \rightarrow (R)^*$

Theorem

(Mathematical Induction on the Type $\texttt{Regex}\,A$*.) Let* $P(r)$ *be a property that pertains to value* r *in data type* $\texttt{Regex}\,A$*. If the four following statements are true:*

$$P(\texttt{Empty}) \text{ is true; and for all } a \in A,\ P(\texttt{Sym}\,a)$$

$$\text{For all } v \in \texttt{Regex}\,A,\ P(v) \text{ implies } P(\texttt{Star}\,v)$$

$$\text{For all } v, w \in \texttt{Regex}\,A,\ P(v) \text{ implies } P(\texttt{Alt}\,v\,w)$$

Then for any $r \in \texttt{Regex}\,A$*,* $P(r)$ *is true.*

Outline

Relations

Definition

The binary, infix, relation $\prec$ is defined by $n \prec n+1$ for all $N \in \mathbb{N}$.

Transitive Closure

The successor constructor (operation) gives rise to (well-ordered) relation $\prec$:

$$\frac{n \in \mathbb{N}}{n \prec n'}$$

Reflexive and transitive closure of an existing (arbitrary) $\prec$ relation.

$$\frac{n \prec m}{n \prec^+ m} \qquad \frac{n \in \mathbb{N}}{n \prec^+ n} \qquad \frac{n \prec^+ m \quad m \prec p}{n \prec^+ p}$$

$$\frac{n \in \mathbb{N}}{n \leq n'} \qquad \frac{n \in \mathbb{N}}{n \leq n} \qquad \frac{n \leq m \quad m \leq p}{n \leq p}$$

Transitive Closure

The transitive closure of a well-founded relation is well-founded.
The reflexive, transitive closure of a relation can never be well-founded. (So we cannot do induction on the relation.)
Hence, the need to do induction on the proof that a pair is in the reflexive, transitive closure of a relation.

References I

Jean H. Gallier.
Logic for Computer Science: Foundations of Automatic Theorem Proving.
Harper & Row, New York, 1986.

James L. Hein.
Discrete Structures, Logic, and Computability.
Jones and Bartlett, Sudbury, Massachusetts, second edition, 2002.

John Clifford Mitchell.
Foundations of Programming Languages.
MIT Press, 1996.