

Computer Supported Modeling and Reasoning

HOL: Arithmetic

Burkhart Wolff

15.01.03

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the [Isabelle/HOL library](#).

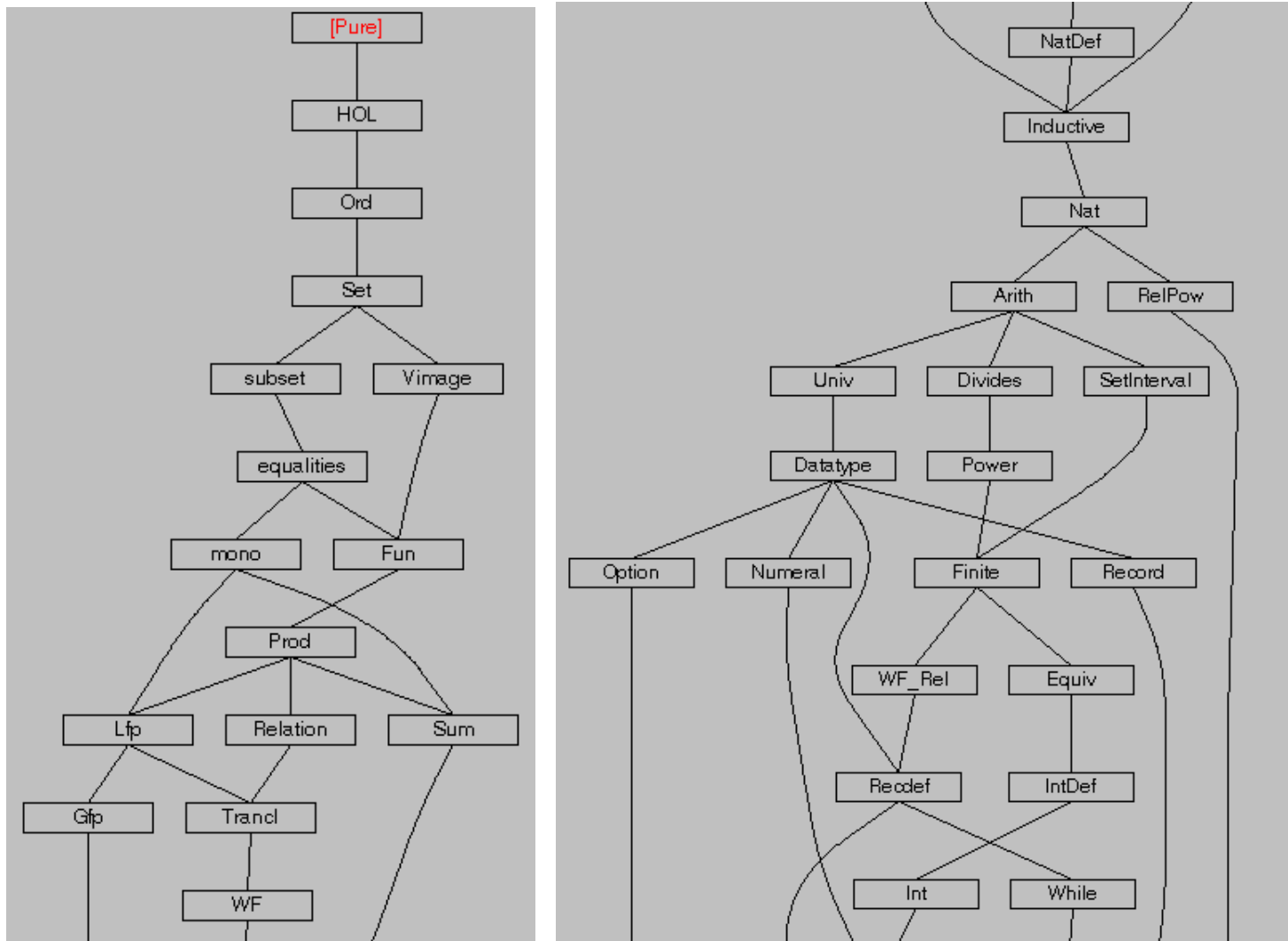
- Orders and sets
- Fixpoints, induction, and recursion
- Arithmetic
- Datatypes

The Roadmap

We are still looking at how the different parts of mathematics are encoded in the [Isabelle/HOL library](#).

- Orders and sets
- Fixpoints, induction, and recursion
- **Arithmetic**
- Datatypes

The Roadmap (in Isabelle/HOL)



Motivation

Current stage of our course:

- On the basis of conservative embeddings, **set theories** can be built safely.
- **Inductive sets** can be reduced to sets and suitably **supported by Isabelle**.
- **Well-founded recursion** can be reduced to sets and supported by Isabelle.

Motivation

Current stage of our course:

- On the basis of conservative embeddings, **set theories** can be built safely.
- **Inductive sets** can be reduced to sets and suitably **supported by Isabelle**.
- **Well-founded recursion** can be reduced to sets and supported by Isabelle.

Can we bridge the gap to computer science?

How can we do arithmetic?

Arithmetic

Which Approach to Take?

- Purely definitional?

Which Approach to Take?

- Purely definitional?

Not possible with eight basic rules (cannot enforce infinity of HOL model)!

Which Approach to Take?

- Purely definitional?
Not possible with eight basic rules (cannot enforce infinity of HOL model)!
- Heavily Axiomatic? I.e., we state natural numbers by Peano axioms and claim analogous axioms for any other number type?

Which Approach to Take?

- Purely definitional?
Not possible with **eight basic rules** (cannot enforce infinity of HOL model)!
- Heavily Axiomatic? I.e., we state natural numbers by **Peano axioms** and claim analogous axioms for any other number type?
Insecure!

Which Approach to Take?

- Purely definitional?
Not possible with **eight basic rules** (cannot enforce infinity of HOL model)!
- Heavily Axiomatic? I.e., we state natural numbers by **Peano axioms** and claim analogous axioms for any other number type?
Insecure!
- Minimally axiomatic? We construct an infinite set, and define numbers etc. as **inductive subset**?

Which Approach to Take?

- Purely definitional?

Not possible with **eight basic rules** (cannot enforce infinity of HOL model)!

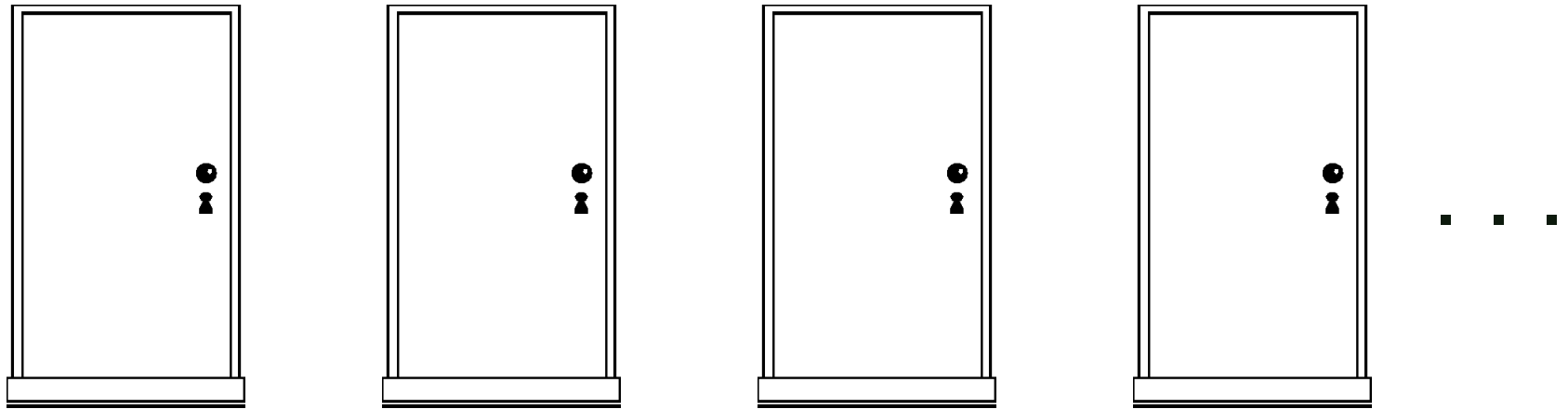
- Heavily Axiomatic? I.e., we state natural numbers by **Peano axioms** and claim analogous axioms for any other number type?

Insecure!

- Minimally axiomatic? We construct an infinite set, and define numbers etc. as **inductive subset**?

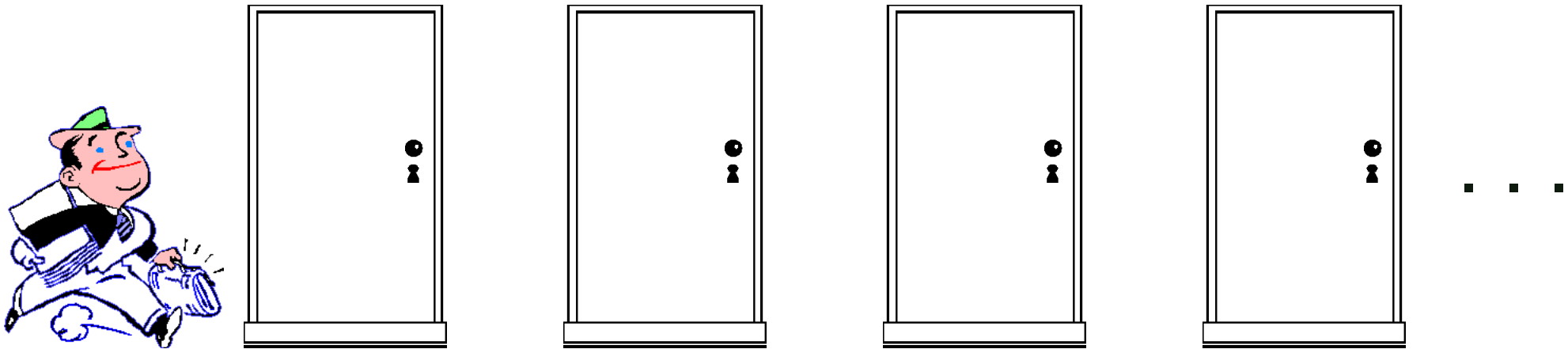
Yes. Finally use **infinity** axiom.

What is Infinity? Cantor's Hotel



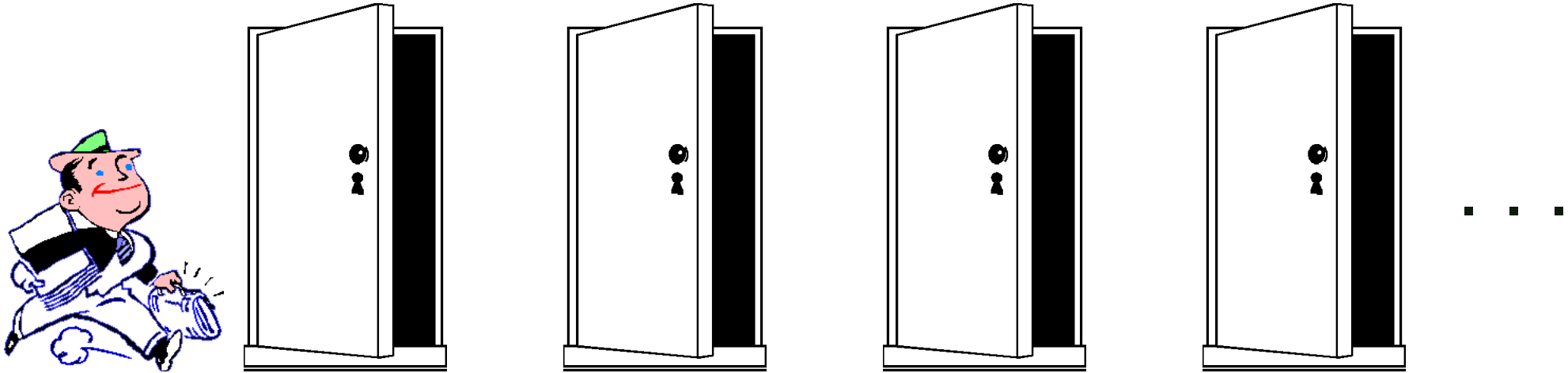
Cantor's hotel has infinitely many rooms.

What is Infinity? Cantor's Hotel



Cantor's hotel has infinitely many rooms. A new guest arrives.

What is Infinity? Cantor's Hotel



Cantor's hotel has infinitely many rooms. A new guest arrives.
The doors open,

What is Infinity? Cantor's Hotel



Cantor's hotel has infinitely many rooms. A new guest arrives. The doors open, and all guests come out of their rooms.

What is Infinity? Cantor's Hotel



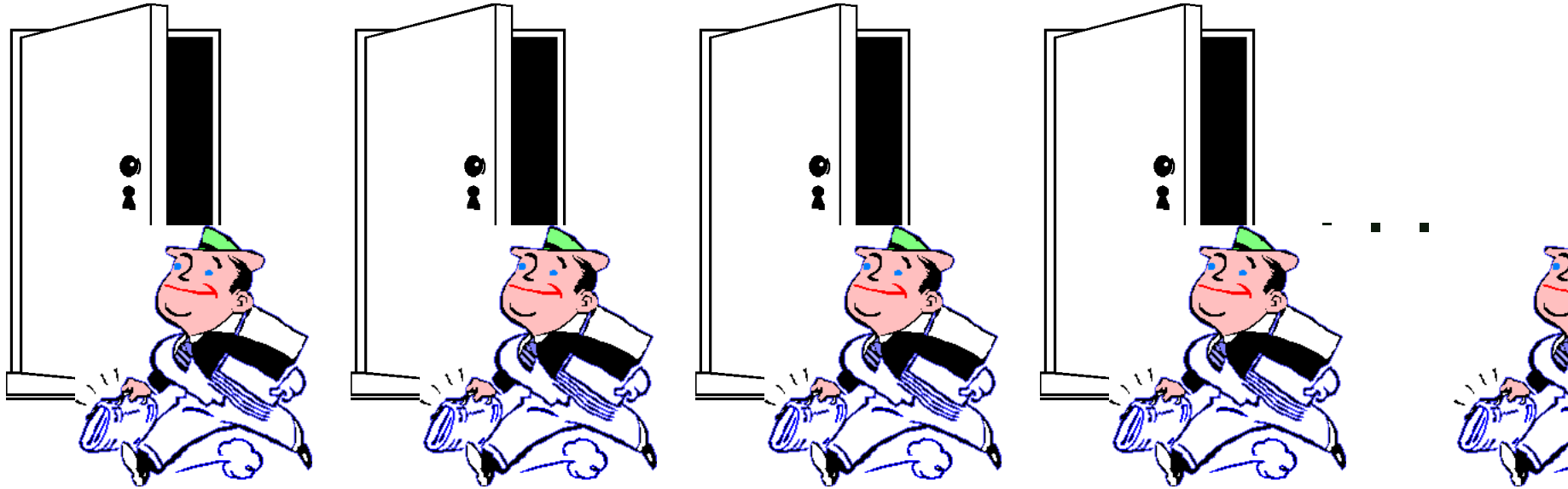
Cantor's hotel has infinitely many rooms. A new guest arrives. The doors open, and all guests come out of their rooms. They move one room forward,

What is Infinity? Cantor's Hotel



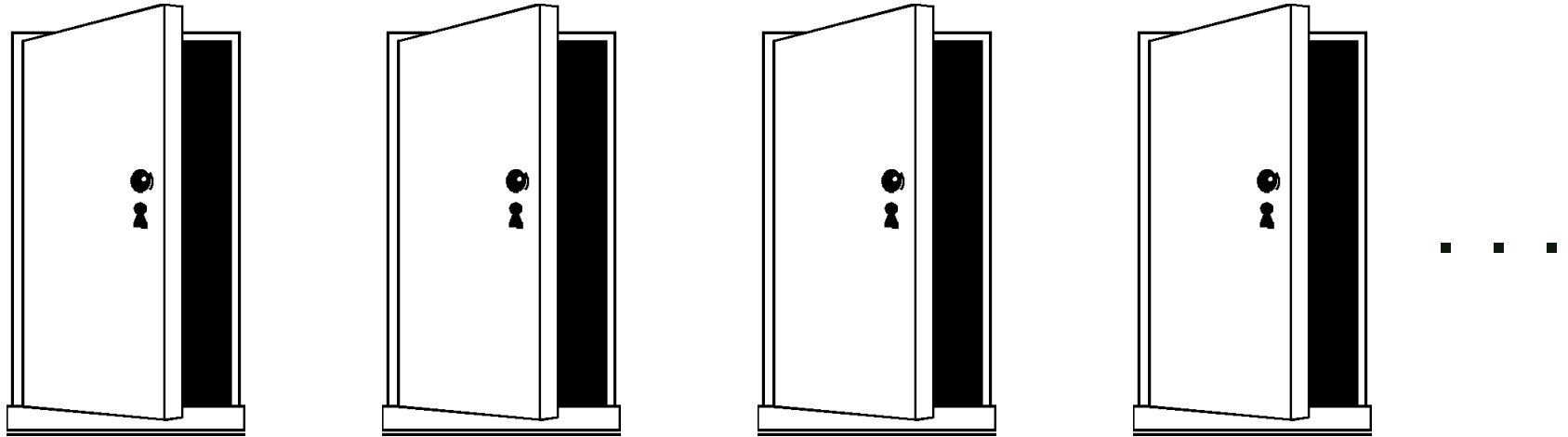
Cantor's hotel has infinitely many rooms. A new guest arrives. The doors open, and all guests come out of their rooms. They move one room forward, the new guest walks towards the first room,

What is Infinity? Cantor's Hotel



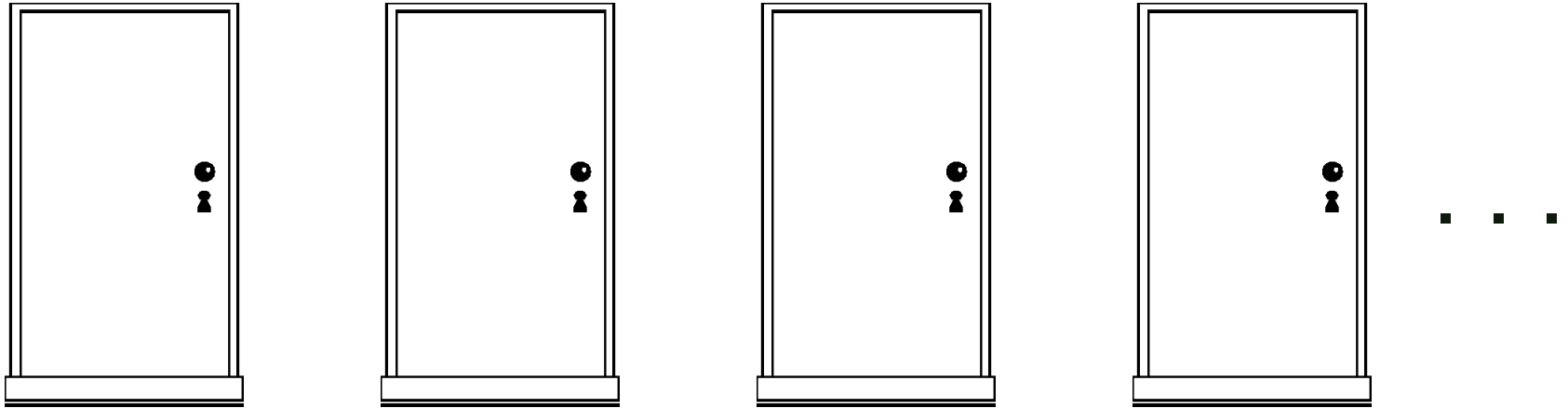
Cantor's hotel has infinitely many rooms. A new guest arrives. The doors open, and all guests come out of their rooms. They **move one room forward**, the new guest walks towards the first room, they turn around,

What is Infinity? Cantor's Hotel



Cantor's hotel has infinitely many rooms. A new guest arrives. The doors open, and all guests come out of their rooms. They **move one room forward**, the new guest walks towards the first room, they turn around, disappear in their new rooms.

What is Infinity? Cantor's Hotel



Cantor's hotel has infinitely many rooms. A new guest arrives. The doors open, and all guests come out of their rooms. They **move one room forward**, the new guest walks towards the first room, they turn around, disappear in their new rooms. The doors close, all guests are accommodated.

Axiom of Infinity

The axiomatic core of datatypes (and hence, numbers):

$$\frac{}{\exists f :: (ind \rightarrow ind).injective f \wedge \neg surjective f} \textit{infty}$$

where

$$\textit{injective } f = \forall xy. f x = f y \rightarrow x = y$$

$$\textit{surjective } f = \forall y. \exists x. y = f x$$

Forces *ind* to be “infinite type” (called “*I*” in [Chu40]).

Type Closed Conservative Extensions

Why must conservative extensions be type-closed? (paying an old debt)

Consider $H = \exists f :: \alpha \Rightarrow \alpha. \text{injective } f \wedge \neg \text{surjective } f$

Type Closed Conservative Extensions

Why must conservative extensions be type-closed? (paying an **old debt**)

Consider $H = \exists f :: \alpha \Rightarrow \alpha. \text{injective } f \wedge \neg \text{surjective } f$

Then the type of H is *bool*, but H contains a subterm of type $\alpha \Rightarrow \alpha$ (H is not type-closed).

Then we could reason as follows ...

Type Closed Conservative Extensions (2)

$(H = \exists f :: \alpha \Rightarrow \alpha.injective\ f \wedge \neg surjective\ f)$

Type Closed Conservative Extensions (2)

$(H = \exists f :: \alpha \Rightarrow \alpha.injective\ f \wedge \neg surjective\ f)$

$H = H$ holds by *refl*
 $\Rightarrow \exists f :: bool \Rightarrow bool.inj\ f \wedge \neg sur\ f =$
 $\quad \exists f :: ind \Rightarrow ind.inj\ f \wedge \neg sur\ f$
 $\Rightarrow False = True$
 $\Rightarrow False$

(unfolding H using two different type instantiations, and then using *Axiom of Infinity* and the fact that there are only finitely many functions on *bool*).

Types Affect the Semantics

Type instantiations may change semantic values, and hence cause **inconsistency**!

Remark: This could have been shown with a simpler example.

Natural Numbers: NatDef.thy

NatDef = Wf +

consts

Zero_Rep :: ind

Suc_Rep :: ind \Rightarrow ind

defs

Suc_Rep_def "Suc_Rep \equiv $\epsilon f :: \text{ind} \Rightarrow \text{ind}. \text{inj}(f)$ "

Zero_Rep_def "Zero_Rep \equiv $\epsilon y. \forall x. \text{Suc_Rep}(x) \neq y$ "

Natural Numbers: NatDef.thy

NatDef = Wf +

consts

Zero_Rep :: ind

Suc_Rep :: ind => ind

defs

Suc_Rep_def "Suc_Rep \equiv $\epsilon f :: \text{ind} \Rightarrow \text{ind}.$ inj (f)"

Zero_Rep_def "Zero_Rep \equiv $\epsilon y. \forall x. \text{Suc_Rep}(x) \neq y$ "

Axiom of Infinity ensures the existence of these choices by the **Hilbert operator**.

inj and *sur* are defined in **Fun.thy**.

Defining *Nat*

Type abstraction:

...

(* type definition *)

typedef (Nat)

nat = "lfp ($\lambda X. \{ \text{Zero_Rep} \} \cup (\text{Suc_Rep}'X)$)" (lfp_def)

Defining *Nat*

Type abstraction:

```
...
(* type definition *)
typedef (Nat)
  nat = "lfp ( $\lambda X. \{Zero\_Rep\} \cup (Suc\_Rep\ 'X)$ )" (lfp_def)
```

Constructor abstraction:

```
...
constdefs
  0      :: nat          "0  $\equiv$  Abs_Nat(Zero_Rep)"
  Suc    :: nat => nat   "Suc  $\equiv$  ( $\lambda n. Abs\_Nat(Suc\_Rep(Rep\_Nat(n)))$ )"
  pred_nat :: (nat * nat) set "pred_nat  $\equiv$  {(m,n). n = Suc m}"
```

Some Theorems in NatDef

$\text{nat_induct} : \llbracket P\ 0; \bigwedge n. P\ n \Rightarrow P\ (\text{Suc}\ n) \rrbracket \Rightarrow P\ n$

$\text{diff_induct} : \llbracket \bigwedge x. P\ x\ 0; \bigwedge y. P\ 0\ (\text{Suc}\ y);$
 $\bigwedge x\ y. P\ x\ y \Rightarrow P\ (\text{Suc}\ x)\ (\text{Suc}\ y) \rrbracket$
 $\Rightarrow P\ m\ n$

Use **least fixpoints** because we want to define a **set**. In particular, `nat_induct` follows (but not “automatically”!) from the **induct theorem** of `Lfp`.

NatDef and Well-founded Orders

`wf_pred_nat`: `wf pred_nat`

`less_linear` : $m < n \vee m = n \vee n < m$

`Suc_less_SucD`: $\text{Suc } m < \text{Suc } n \Rightarrow m < n$

...

These **theorems** involve **well-founded orders**.

Using Primitive Recursion

`Nat.thy` defines rich arithmetic theory on natural numbers.

primrec

add_0 "0 + n = n"

add_Suc "Suc m + n = Suc(m + n)"

primrec

diff_0 "m - 0 = m"

diff_Suc "m - Suc n = (case m - n of 0 => 0 | Suc k => k)"

primrec

mult_0 "0 * n = 0"

mult_Suc "Suc m * n = n + (m * n)"

Using Primitive Recursion

`Nat.thy` defines rich arithmetic theory on natural numbers.

`primrec`

`add_0` " $0 + n = n$ "

`add_Suc` " $\text{Suc } m + n = \text{Suc}(m + n)$ "

`primrec`

`diff_0` " $m - 0 = m$ "

`diff_Suc` " $m - \text{Suc } n = (\text{case } m - n \text{ of } 0 \Rightarrow 0 \mid \text{Suc } k \Rightarrow k)$ "

`primrec`

`mult_0` " $0 * n = 0$ "

`mult_Suc` " $\text{Suc } m * n = n + (m * n)$ "

Uses special `primrec` syntax for defining recursive functions.

Also uses `case` construct.

Some Theorems in Nat

$$\text{add_0_right : } m + 0 = m$$

$$\begin{aligned} \text{add_ac: } & m + n + k = m + (n + k) \\ & m + n = n + m \\ & x + (y + z) = y + (x + z) \end{aligned}$$

$$\begin{aligned} \text{mult_ac: } & m * n * k = m * (n * k) \\ & m * n = n * m \\ & x * (y * z) = y * (x * z) \end{aligned}$$

Note **third part** of `add_ac`, `mult_ac`, respectively.

Technically, `add_ac` and `mult_ac` are lists of **thm**'s.

Integers

The integers are implemented as equivalence classes over $\text{nat} \times \text{nat}$.

IntDef = Equiv + Arith +

constdefs

intrel :: " $((\text{nat} * \text{nat}) * (\text{nat} * \text{nat})) \text{ set}$ "

"intrel $\equiv \{p. \exists x1 y1 x2 y2.$

$p = ((x1::\text{nat}, y1), (x2, y2)) \wedge x1 + y2 = x2 + y1\}$ "

typedef (Integ)

int = "UNIV//intrel" (Equiv. quotient_def)

Some Theorems in IntArith

$$\text{zminus_zadd_distrib: } -(z + w) = -z + -w$$

$$\text{zminus_zminus: } -(-z) = z$$

$$\text{zadd_ac: } z1 + z2 + z3 = z1 + (z2 + z3)$$

$$z + w = w + z$$

$$x + (y + z) = y + (x + z)$$

$$\text{zmult_ac: } z1 * z2 * z3 = z1 * (z2 * z3)$$

$$z * w = w * z$$

$$z1 * (z2 * z3) = z2 * (z1 * z3)$$

Further Number Theories

- Binary Integers (`Bin.thy`, for fast computation)
- Rational Numbers (`HOL-Real/PRat.thy`)
- Reals (`HOL-Real/PReal.thy`; based on Dedekind-cuts of rationals [[Ded](#)])
- Hyperreals (`HOL-Real/RealDef.thy` for non-standard analysis)
- Machine numbers; see work for [Intel's PentiumIV](#); built in HOL-light [[Har98](#), [Har00](#)]

Conclusion

Although building up a library in a **definitional way**, we can build

- the **naturals** (as **type definition** based on *ind*), and

Conclusion

- Although building up a library in a **definitional way**, we can build
- the **naturals** (as **type definition** based on *ind*), and
 - **higher number theories** in HOL (via equivalence construction).

Conclusion

Although building up a library in a **definitional way**, we can build

- the **naturals** (as **type definition** based on *ind*), and
- **higher number theories** in HOL (via equivalence construction).

Potential for

- analysis of processor arithmetic units,

Conclusion

Although building up a library in a **definitional way**, we can build

- the **naturals** (as **type definition** based on *ind*), and
- **higher number theories** in HOL (via equivalence construction).

Potential for

- analysis of processor arithmetic units, and
- function analysis in HOL (combination with computer algebra systems such as Mathematica).

Conclusion

Although building up a library in a **definitional way**, we can build

- the **naturals** (as **type definition** based on *ind*), and
- **higher number theories** in HOL (via equivalence construction).

Potential for

- analysis of processor arithmetic units, and
- function analysis in HOL (combination with computer algebra systems such as Mathematica).

Future: analysis of hybrid systems.

Conclusion

Although building up a library in a **definitional way**, we can build

- the **naturals** (as **type definition** based on *ind*), and
- **higher number theories** in HOL (via equivalence construction).

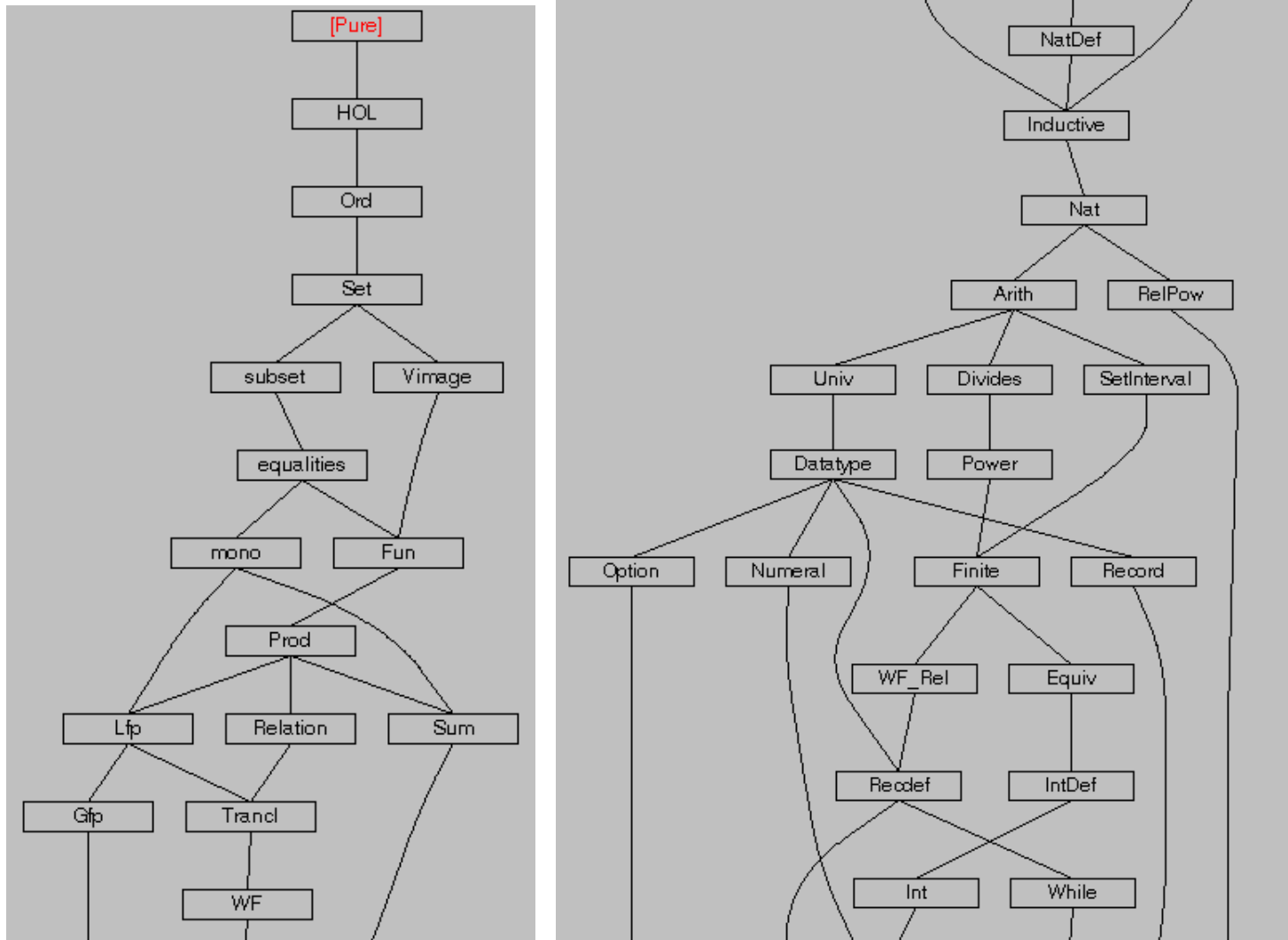
Potential for

- analysis of processor arithmetic units, and
- function analysis in HOL (combination with computer algebra systems such as Mathematica).

Future: analysis of hybrid systems.

The methodological overhead can be tackled by powerful mechanical support, since many proof-tasks are routine.

The Roadmap (in Isabelle/HOL)



References

- [Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
- [Ded] Dedekind. *Cuts of Rationals*. ??, ??
- [Har98] John Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998.
- [Har00] John Harrison. Formal verification of the ia/64 division algorithms. In Mark Aagaard and John Harrison, editors, *Proceedings of the 13th TPHOLs*, volume 1869 of *LNCS*, pages 233–251. Springer-Verlag, 2000.

More Detailed Explanations

Well-founded Recursion and Sets

What's the role of sets in defining well-founded recursion? It's the fact that well-founded recursion is based on well-founded orders, which are [relations](#), which are sets.

Enforcing Infinity

Our intuition/knowledge about arithmetics clearly requires that there are infinite sets, e.g., the set of infinite numbers. Technically, the HOL model of the set of natural numbers must be an infinite set, otherwise we would not be willing to say that have “modeled” arithmetic.

The Peano Axioms

The Peano axioms include

- $0 \in \mathit{nat}$
- $\forall x. x \in \mathit{nat} \rightarrow \mathit{Suc}(x) \in \mathit{nat}$
- $\forall x. \mathit{Suc}(x) \neq 0$
- $\forall P. (P(0) \wedge \forall n. (P(n) \rightarrow P(\mathit{Suc}(n)))) \rightarrow \forall n. P(n)$

Successors on Rooms

This means, there must be a successor function on rooms. To each room, it assigns the “next” room.

injective **and** *surjective*

These constants (actually called *inj* and *sur*) are defined in [Fun.thy](#).

Is it Necessary to Add an Axiom?

Note that theoretically, it is not needed to add the infinity axiom (or some **equivalent formulation**) to HOL. Instead one could add the infinity axiom as premise to each arithmetic theorem that one wants to prove.

However this would not be a viable approach since the resulting formulas would be very, very complicated.

Numbers as Datatype

The natural numbers can be built as an algebraic datatype by having a constant 0 and a term constructor Suc (for **successor**).

inj and *sur*

We use *inj* and *sur* as abbreviations for *injective* and *surjective*.

NatDef.thy

This file should be contained in your Isabelle distribution. In Freiburg it is `/usr/local/isabelle/Isabelle/src/HOL/NatDef.thy`

NatDef .ML

This file should be contained in your Isabelle distribution. In Freiburg it is `/usr/local/isabelle/Isabelle/src/HOL/NatDef.ML`

Existence of Choices

Recall that $\epsilon x.P x$ returns an a such that $P a$ holds, provided such an a exists.

The [Axiom of Infinity](#) states that here, the choices do exist.

Note that if these choices did not exist, it would [not mean that](#) `Suc_Rep` and `Zero_Rep` are undefined. But they simply would not have the required properties, which we now show.

Since they do exist, it is easy to show

$$\begin{aligned} & \text{inj}(\text{Suc_Rep}) \\ & \forall x. \text{Suc_Rep}(x) = \text{Zero_Rep} \end{aligned}$$

(the rule [selectI](#) is crucial here). Incidentally, this corresponds to the way infinity is axiomatized in Isabelle. In [NatDef.thy](#), we find:

rules

(*the axiom of infinity in 2 parts*)

inj_Suc_Rep "inj (Suc_Rep)"

Suc_Rep_not_Zero_Rep "Suc_Rep(x) \neq Zero_Rep"

A Type Definition Based on an Inductive Set

Note the two ingredients for defining the type `nat`:

- An **inductively defined set** `Nat`, i.e., a set defined as fixpoint of a monotone function. In Isabelle (`NatDef.thy`), the **inductive syntax** is used for this purpose. This **automatically generates an induction rule** for the set.
- A **type definition** based on this set, defined using the **typedef syntax**.
Recall that this process automatically generates the two constants **`Abs_Nat`** and **`Rep_Nat`**.

But note: the induction theorem is not inherited automatically. More precisely, the `typedef` syntax does not cause the type `nat` to inherit the inductive theorem of the set `Nat`. The theorem `nat_induct` is explicitly proven in `NatDef.ML`.

Constructor Abstraction

Based on the generic constants `Abs_Nat` and `Rep_Nat`, we define all the constants that we need to work conveniently with `nat`, most importantly, `0` and `Suc`.

Nat.thy

This file should be contained in your Isabelle distribution. In Freiburg it is `/usr/local/isabelle/Isabelle/src/HOL/Nat.thy`

The case Statement for nat

The case statement for `nat` is a function of type $\text{nat} \Rightarrow (\text{nat} \Rightarrow \text{nat}) \Rightarrow \text{nat} \Rightarrow \text{nat}$. `case z f n` is defined as follows (using a common mathematical notation):

$$\text{case } z \ f \ n = \begin{cases} z & \text{if } n = 0 \\ f \ k & \text{if } n = \text{Suc } k \end{cases}$$

This syntax

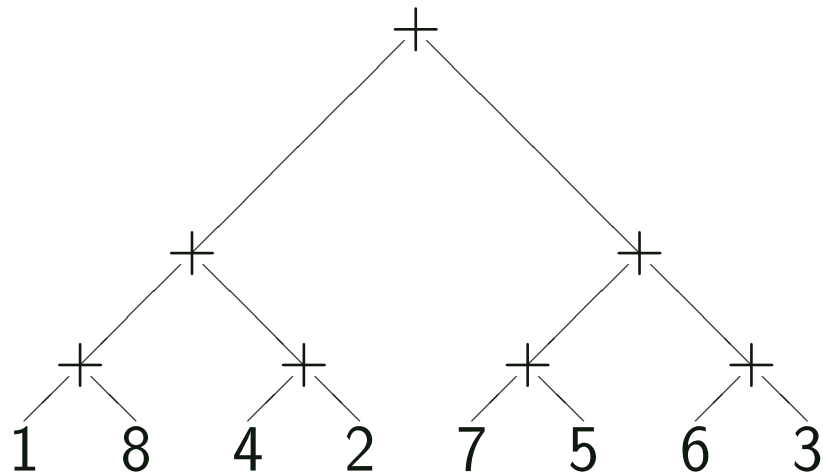
`diff_Suc "m - Suc n = (case m - n of 0 => 0 | Suc k => k)"`

used on the slide is a paraphrasing of the original syntax. In the original syntax it would read `case 0 ($\lambda x.x$) (n - m)`.

Left Commutation

The theorems $x + (y + z) = y + (x + z)$ and $x * (y * z) = y * (x * z)$ are called **left-commutation laws** and are crucial for **(ordered) rewriting**.

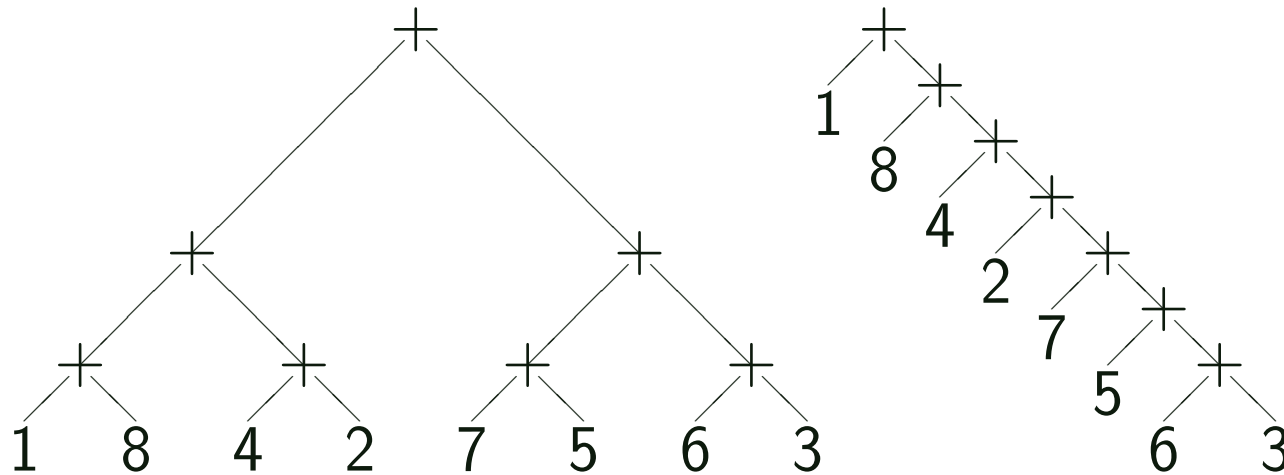
Suppose we have the term shown below.



Left Commutation

The theorems $x + (y + z) = y + (x + z)$ and $x * (y * z) = y * (x * z)$ are called **left-commutation laws** and are crucial for **(ordered) rewriting**.

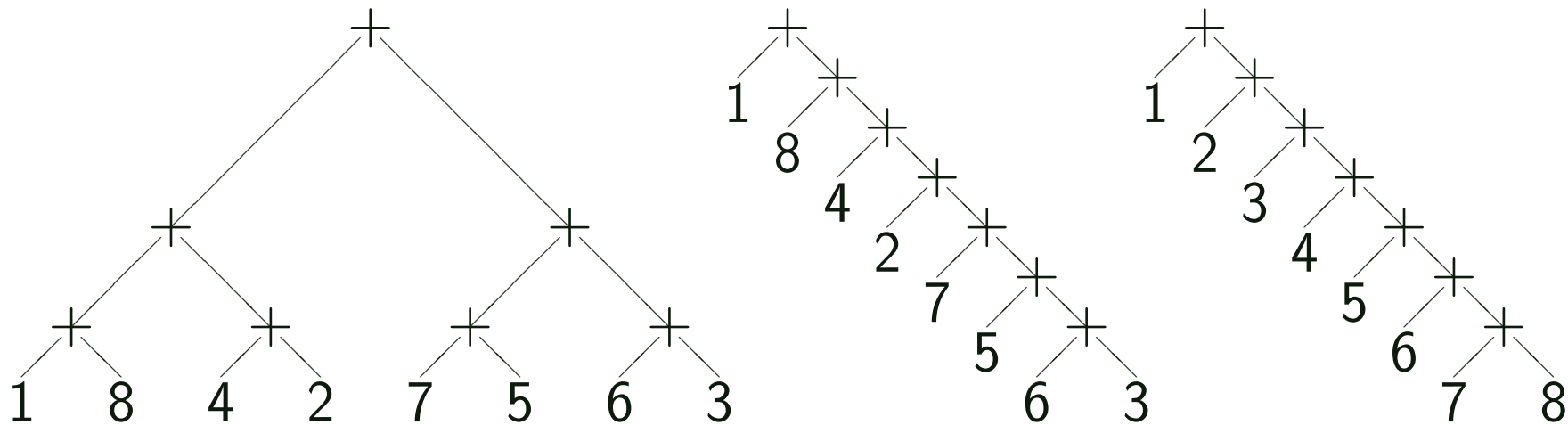
Suppose we have the term shown below. Using associativity ($m + n + k = m + (n + k)$) this will be rewritten to the second term.



Left Commutation

The theorems $x + (y + z) = y + (x + z)$ and $x * (y * z) = y * (x * z)$ are called **left-commutation laws** and are crucial for **(ordered) rewriting**.

Suppose we have the term shown below. Using associativity ($m + n + k = m + (n + k)$) this will be rewritten to the second term. Using left-commutation, this will be rewritten to the third term. This is a so-called **AC-normal form**, for an appropriately chosen **term ordering**.



equivalence classes

Recall the general concept of an **equivalence relation**. Generally, for a set S and an equivalence relation R defined on the set, one can define $S//R$, the **quotient of S w.r.t. R** .

$$S//R = \{A \mid A \subseteq S \wedge \forall x, y \in A. (x, y) \in R\}$$

That is, one partitions the set S into subsets such that each subset collects equivalent elements. This is a standard mathematical concept.

We do not go into the Isabelle details here, but we explain how this works for the integers. One can view a pair (n, m) of natural numbers as representation of the integer $n - m$. But then (n, m) and (n', m') represent the same integer if and only if $n - m = n' - m'$, or equivalently, $n + m' = n' + m$. In this case (n, m) and (n', m') are said to be **equivalent**.

The construction of the integer type is based on this equivalence relation, called `intrel`. More precisely, the definition of the integers will be based on the set of all pairs of naturals (which corresponds to the `UNIV` constant on the type `nat × nat`) modulo the equivalence `intrel`. In other words, it will be based on the quotient of the set of pairs of naturals w.r.t. `intrel`.