The Language

- **Individual variables**: $a, b, c, f, g, h, m, n, x, y, z, \ldots$

- Set of *individual constants*, containing at least:
  - $k, s$ (combinators), $p, p_0, p_1$ (pairing and projections), $0$ (zero), $s_N$ (successor), $p_N$ (predecessor) and $d_N$ (definition by numerical cases).

- **Terms** ($r, s, t, \ldots$) are built up from the variables and constants by means of the function symbol $\cdot$ for (partial) application.
  - We use $(st)$ or $st$ as an abbreviation for $(s \cdot t)$ and adopt the convention of association to the left, i.e. $s_1s_2 \ldots s_n$ stands for $(\ldots (s_1 \cdot s_2) \ldots s_n)$.

- The **atomic formulas** are $s\downarrow$, $N(s)$ and $s = t$.
  - Since we work with a logic of partial terms, it is not guaranteed that all terms have values, and $s\downarrow$ has to be read as *s is defined* or *s has a value*.
  - $N(s)$ says that $s$ is a natural number.

- The **formulas** $(A, B, C, \ldots, \varphi, \psi, \chi, \ldots)$ are generated from the atomic formulas by closing against the usual propositional connectives and quantifiers.
The Logic of Partial Terms

We use the following abbreviation: \( t \simeq s :\Leftrightarrow t \downarrow \lor s \downarrow \rightarrow t = s. \)

- A complete axiomatization of the classical propositional logic.
- Quantifier axioms
  
  (Q1) \( (\forall x. \varphi) \land t \downarrow \rightarrow \varphi[t/x], \)
  
  (Q2) \( \varphi[t/x] \land t \downarrow \rightarrow \exists x. \varphi. \)

- Equality axioms
  
  (E1) \( x = x, \)
  
  (E2) \( t \simeq s \land \varphi(t) \rightarrow \varphi(s). \)

- Strictness axioms
  
  (S1) \( R(t_1, \ldots, t_n) \rightarrow t_1 \downarrow \land \ldots \land t_n \downarrow, \) if \( R \) is a \( n \)-ary relation symbol,
  
  (S2) \( ts \downarrow \rightarrow t \downarrow \land s \downarrow, \)
  
  (S3) \( c \downarrow, \) if \( c \) is a constant,
  
  (S4) \( x \downarrow. \)

As rules we have \textit{Modus ponens} and the \textit{usual} quantifier rules.
The basic theory of operations and numbers

I. Partial combinatory algebra
   (1) \( k x y = x \),
   (2) \( s x y \downarrow \land s x y z \simeq x z (y z) \).

II. Pairing and projection
   (3) \( p_0 x \downarrow \land p_1 x \downarrow \),
   (4) \( p_0 (p x y) = x \land p_1 (p x y) = y \).

III. Natural numbers
   (5) \( N(0) \land \forall x. N(x) \rightarrow N(s_N x) \),
   (6) \( \forall x. N(x) \rightarrow s_N x \neq 0 \land p_N (s_N x) = x \),
   (7) \( \forall x. N(x) \land x \neq 0 \rightarrow N(p_N x) \land s_N (p_N x) = x \).

IV. Definition by cases on \( N \)
   (8) \( N(v) \land N(w) \land v = w \rightarrow d_N x y v w = x \),
   (9) \( N(v) \land N(w) \land v \neq w \rightarrow d_N x y v w = y \).
**Theorem**  For every variable \( x \) and every term \( t \), there exists a term \( \lambda x.t \) whose free variables are those of \( t \), excluding \( x \), such that

\[
\text{BON} \vdash \lambda x.t \downarrow \land (\lambda x.t) \cdot x \simeq t \quad \text{and} \quad \text{BON} \vdash s \downarrow \rightarrow (\lambda x.t) \cdot s \simeq t[s/x].
\]

\((\lambda x.t)\) is inductively defined as follows:

\[
\begin{align*}
\lambda x.x & := s \cdot k \cdot k, & \hat{\lambda} x.x & := s \cdot k \cdot k, \\
\lambda x.y & := k \cdot y, \quad \text{if } x \not\equiv y, & \hat{\lambda} x.t & := k \cdot t, \quad \text{if } x \not\in \text{FV}(t), \\
\lambda x.c & := k \cdot c, \quad \text{if } c \text{ is a constant}, \\
\lambda x.r s & := s \cdot (\lambda x.r) \cdot (\lambda x.s) & \hat{\lambda} x.r s & := s \cdot (\hat{\lambda} x.r) \cdot (\hat{\lambda} x.s).
\end{align*}
\]

This definition ensures \( \lambda x.t \downarrow \) even for \( \neg t \downarrow \). However, we would have \( \neg \hat{\lambda} x.t \downarrow \) because of strictness.

**Remark**  In contrast to the definition of \( \hat{\lambda} \) — and ordinary \( \lambda \) calculus — \( \lambda \) is not compatible with substitution:

For \( x \not\equiv y \) the equality \((\lambda x.t)[s/y] = \lambda x.t[s/y]\) does not hold in general.
Recursion Theorem

**Theorem**  There exists a term $\text{rec}$ such that

$$\text{BON \vdash rec} \, f \downarrow \land \forall x. \text{rec} \, f \, x \simeq f \, (\text{rec} \, f) \, x.$$ 

$$\text{rec} := \lambda g. (\lambda y, z. g \, (y \, y) \, z) \, (\lambda y, z. g \, (y \, y) \, z).$$

$$\text{rec} \, f \simeq (\lambda g. (\lambda y, z. g \, (y \, y) \, z) \, (\lambda y, z. g \, (y \, y) \, z)) \, f$$
$$\simeq (\lambda y, z. f \, (y \, y) \, z) \, (\lambda y, z. f \, (y \, y) \, z)$$

$$\text{rec} \, f \, x \simeq (\lambda y, z. f \, (y \, y) \, z) \, (\lambda y, z. f \, (y \, y) \, z) \, x$$
$$\simeq f \, ((\lambda y, z. f \, (y \, y) \, z) \, (\lambda y, z. f \, (y \, y) \, z)) \, x$$
$$\simeq f \, (\text{rec} \, f) \, x$$

**Remark**  In the total setting one can define a term $\text{rec}'$ such that $\text{rec}' \, f = f \, (\text{rec}' \, f)$. In the partial setting the recursion equation holds only pointwise.
Solving recursions

\( f \ x \simeq t[f, x] \) can be solved by \( f := \text{rec} (\lambda g, x.t[g, x]) \).

\[
\begin{align*}
  f \ x & \simeq \text{rec} (\lambda g, x.t[g, x]) x \\
  & \simeq (\lambda g, x.t[g, x]) (\text{rec} (\lambda g, x.t[g, x])) x \\
  & \simeq (\lambda g, x.t[g, x]) f x \\
  & \simeq t[f, x]
\end{align*}
\]

Example

\[
x + y = \begin{cases} 
  x & \text{if } y = 0, \\
  (x + (y - 1)) + 1 & \text{if } y \neq 0.
\end{cases}
\]

\[
\text{plus } x \ y = d_s \ x \ (s_N (\text{plus } x (p_N y))) \ y \ 0
\]

\[
\text{plus} := \text{rec} (\lambda f, x, y.d_s \ x \ (s_N (f \ x (p_N y)))) \ y \ 0)
\]

Question: What about “Totality”:
\( \forall n, m \in N. (\text{plus } n \ m) \in N? \)
Non-strict case distinction

From Strictness follows

\[ d_N r s u v \downarrow \rightarrow r \downarrow \land s \downarrow \]

To define a (recursive) function by case definition, we introduce the following “strong” definition by cases:

\[ d_s r s u v := d_N (\lambda z. r) (\lambda z. s) u v 0 \]

where the variable \( z \) does not occur in the terms \( r \) and \( s \).
Induction

\[(L-I_N) \quad \varphi(0) \land (\forall x \in \mathbb{N}. \varphi(x) \rightarrow \varphi(s_N x)) \rightarrow \forall x \in \mathbb{N}. \varphi(x)\]

\[(L-I_N)\] can be used to prove totality of a “function” \( f \) on \( \mathbb{N} \):

\[\forall x \in \mathbb{N}. f x \in \mathbb{N}.\]

It is immediate that Peano Arithmetic PA can be embedded in BON + \((L-I_N)\).
BON + (\mathcal{L} \text{-} \text{I}_\mathbb{N}) has a natural recursion-theoretic interpretation, where the term universe is interpreted by the natural numbers and the application $a \cdot b$ is translated into $\{a\}(b)$ ($\{n\}$ for $n = 0, 1, 2, 3, \ldots$ is a standard enumeration of the partial recursive functions).

**Theorem** BON + (\mathcal{L} \text{-} \text{I}_\mathbb{N}) and PA are proof-theoretic equivalent.

Another class of models are given by *term models*. The universe consists of all (closed) terms of the language; a reduction relation is defined based on rewrite rules for the combinators; equality is defined as “having a common reduct”; Natural numbers are those terms which reduce to a numeral. Existence can be interpreted either as “having a normal form”; or it can be trivialized by taking the full universe.
Relation to Computer Science

Applicative Theories provide a framework to prove directly properties — such as: correctness, termination, etc. — of (the pure functional core) of Functional programming languages.

There are two main distinctions for functional programming languages: typed vs. type-free and strict vs. lazy. Applicative Theories follow the type-free and strict paradigm. SCHEME is an example of a functional programming language for which applicative theories can be used.

Types are not build in in the definition of the language (which would result in syntactactic restrictions), but represented as predicates, for example \( \mathbb{N} \). That a term “has” a certain type has to be proven within the theory.

- Applicative theories allow to introduce new “datatypes” by adding new constants and theirs characteristic axioms, without any coding.
- The partial logic allows to speak about termination within the language.
The language of Explicit Mathematics is as for BON, but including additionally

- individual constants $c_e, e \in \mathbb{N}$ (elementary comprehension) and $j$ (join),
- Type variables: $X, Y, Z, \ldots$,
- $=$ is used also on the type level,
- Two binary relation symbols $\in$ (membership) and $\mathcal{R}$ (naming; representation); their first arguments are (individual) terms, the second arguments types.

The formulas are build as usual, from the extended set of atomic formulae, including quantification over type variables: $\exists X. \varphi$ and $\forall X. \varphi$.

$t \in X$ is read $t$ belongs to $X$;

$\mathcal{R}(t, X)$ is read $t$ is a name of $X$. 
Ontological axioms

- Every type has a name: $\exists x. \mathcal{R}(x, U)$,
- One name names only one type: $\mathcal{R}(a, U) \land \mathcal{R}(a, V) \rightarrow U = V$,
  (but one type can have different names)
- Types are extensional: $(\forall x. x \in U \leftrightarrow x \in U) \rightarrow U = V$.

Elementary Comprehension

An formula $\varphi$ is called elementary if it contains neither the relation symbol $\mathcal{R}$ nor bound type variables.

Let $\varphi(x, \vec{y}, \vec{Z})$ be an elementary formula with Gödel number $m = \ulcorner \varphi(x, \vec{y}, \vec{Z}) \urcorner$

- $\exists X. \forall x. x \in X \leftrightarrow \varphi(x, \vec{y}, \vec{Z})$,
- $\mathcal{R}(\vec{z}, \vec{Z}) \land (\forall x. x \in X \leftrightarrow \varphi(x, \vec{y}, \vec{Z})) \rightarrow \mathcal{R}(c_m \vec{y} \vec{z}, X)$.

We write $t \in \{ x \mid \varphi(x, \vec{y}, \vec{Z}) \}$ for $\forall X. \mathcal{R}(\vec{z}, \vec{Z}) \land \mathcal{R}(c_m \vec{y} \vec{z}, X) \rightarrow t \in X$ where $m = \ulcorner \varphi(x, \vec{y}, \vec{Z}) \urcorner$. 
EET allows the straightforward definition of (names of) several useful types:

- **Nat**: 
  \[ \{ x \mid N(x) \} \]
  \[ t \in \text{Nat} \leftrightarrow (\forall X. \mathcal{R}(c_n, X) \rightarrow t \in X) \text{ where } n = \upharpoonright N(x) \upharpoonleft \]
  \[ t \in \text{Nat} \leftrightarrow (\forall X. \mathcal{R}(c_n, X) \rightarrow t \in X \land (\forall s. s \in X \leftrightarrow N(s))) \]
  \[ t \in \text{Nat} \leftrightarrow N(t) \]

- **X ∪ Y**: 
  \[ \{ x \mid x \in X \land x \in Y \} \]

- **X → Y**: 
  \[ \{ f \mid \forall x \in X. f \, x \in Y \} \]

- **X ↾ Y**: 
  \[ \{ f \mid \forall x \in X. f \, x \downarrow \rightarrow f \, x \in Y \} \]

- **V**: 
  \[ \{ x \mid x = x \} \]

The restriction to elementary formulae is essential, to avoid Russell-like types:

- **R**: 
  \[ \{ x \mid \exists X. \mathcal{R}(x, X) \land \neg (x \in X) \} \]

- Names can be interpreted by *characteristic functions* (algorithms) of the types.

- While the types are extensional, the names are intensional.
Join

\[ \mathcal{R}(t) := \exists X. \mathcal{R}(t, X) \]
\[ s \in t := \exists X. \mathcal{R}(t, X) \land s \in X \]

Join

- \[ \mathcal{R}(a, X) \land (\forall x \in X. \mathcal{R}(f x)) \rightarrow \mathcal{R}(j a f) \land \Sigma(a, f, j a f) \]

In this axiom the formula \( \Sigma(a, f, b) \) expresses that \( b \) names the disjoint union of \( f \) over \( a \), i.e.

\[ \Sigma(a, f, b) := \forall x. x \in b \leftrightarrow \exists y, z. x = p y z \land y \in a \land z \in f y. \]
Inductive Generation

\[
\text{Closed}(a, b, S) := (\forall x \in a)[(\forall y \in a)((y, x) \in b \rightarrow y \in S) \rightarrow x \in S].
\]

Consider \( b \) as the code of a binary relation. Then this definition means that \( S \) is a type which contains a \( c \in a \) if all predecessors of \( c \) in \( a \) with respect to \( b \) belong to \( S \).

*Inductive generation* is now given by the following axioms

\[
\mathcal{R}(a) \wedge \mathcal{R}(b) \rightarrow \exists X. \mathcal{R}(i(a, b), X) \wedge \text{Closed}(a, b, X),
\]

\[
\mathcal{R}(a) \wedge \mathcal{R}(b) \wedge \text{Closed}(a, b, \varphi) \rightarrow \forall x \in i(a, b). \varphi(x)
\]

for all formulas \( \varphi(u) \). Thus inductive generation states the existence of accessible parts (uniform in the corresponding names)
Feferman’s theory \( T_0 \)

\[ T_0 = \text{EET} + (\text{Join}) + (\text{IG}) + (\text{F-I}_\text{N}) \]

Theories of Explicit Mathematics play an important role in the proof-theoretic analysis of subsystems of second-order arithmetic; in particular, \( T_0 \) was used in the ordinal analysis of \((\Delta^1_2 - \text{CA}) + (\text{BI})\).
**Least fixed points**

**Example** \( f(x) = f(x + 1) \)

Obviously, every constant function is a solution of this equation:

\[
\begin{align*}
f_0 &= \lambda x.0, \\
f_1 &= \lambda x.1, \\
f_2 &= \lambda x.2, \\
&\ldots
\end{align*}
\]

However, the *least* solution — with respect to the definedness-order — is the total undefined function, i.e. a function \( f_\bot \) such that \( \forall x. \neg f_\bot(x) \downarrow \)

*All programming languages allowing to define recursive functions will give this solution!*  

\[
\text{rec} \left( \lambda f, x.f(x + 1) \right)
\]

yields just a fixed point of this functional equation, i.e. all we can prove about it is:

\[
\begin{align*}
\left( \text{rec} \left( \lambda f, x.f(x + 1) \right) \right) x &\simeq \left( \lambda f, x.f(x + 1) \right) \left( \text{rec} \left( \lambda f, x.f(x + 1) \right) \right) x \\
&\simeq \left( \text{rec} \left( \lambda f, x.f(x + 1) \right) \right) (x + 1)
\end{align*}
\]

We can not prove (or disprove) that \( \text{rec} \left( \lambda f, x.f(x + 1) \right) \) is (extensional) equivalent with \( f_0, f_1, f_2, \ldots \) or \( f_\bot \).
The theory LFP

- Everything is a number
  \[ \forall x. N(x). \]

- Computability
  (Comp.1) \[ \forall f \forall x \forall n \in N. c(f, x, n) = 0 \vee c(f, x, n) = 1, \]
  (Comp.2) \[ \forall f \forall x. f x \downarrow \iff (\exists n \in N. c(f, x, n) = 0). \]

  \[ c(f, x, n) = 0 \] expresses that the computation of \( f x \) has terminated after \( n \) steps.

  \[ \text{LFP := BON} + (\text{Comp}) + \forall x. N(x) + (\mathcal{L} \cdot \text{I}_N). \]

Remark All new axioms of LFP are compatible with the recursion-theoretic model (use Kleene’s \( T \)-predicate to verify the computability axioms). They do not exceed the strength of Peano Arithmetic.
Lemma

1. There exists a closed term $\text{not}_N$ so that LFP proves $\neg \text{N}(\text{not}_N)$.
2. There exists a closed term $b$ so that LFP proves $\forall x. \neg b \ x \downarrow$.

The first assertion holds already in BON:

$$\text{not}_N := \text{rec} (\lambda f, x. d_N 1 0 (f x) 0) 0.$$  

or $\text{not}_N := \text{rec} \ g$ with:

$$g \ x \ \simeq \ \begin{cases} 
1 & \text{if } g \ x = 0, \\
0 & \text{if } g \ x \in \mathbb{N} \land g \ x \neq 0.
\end{cases}$$

With the axiom $\forall x.\text{N}(x)$ in LFP it follows that $\neg \text{not}_N \downarrow$ (which is not provable in BON!).

So for the second assertion we set:

$$b := \lambda x. \text{not}_N$$
Classes

A formula $A$ containing exactly $x$ as free variable will be called a class. Let $A$ and $B$ be classes and let $\varphi$ be an arbitrary formula.

$$
t \in A := t \downarrow \land A[t/x],
$$

$$
A \rightarrow B := \forall y. y \in A \rightarrow xy \in B,
$$

$$
A \bowtie B := \forall y. y \in A \land xy \downarrow \rightarrow xy \in B,
$$

$$
r \sqsubseteq s := r \downarrow \rightarrow r = s,
$$

$$
f \sqsubseteq_{A \bowtie B} g := \forall x \in A. fx \sqsubseteq gx,
$$

$$
f \cong_{A \bowtie B} g := f \sqsubseteq_{A \bowtie B} g \land g \sqsubseteq_{A \bowtie B} f.
$$

$r \sqsubseteq s$ says that if $r$ has a value, then $r$ is equal to $s$.

$f \sqsubseteq_{A \bowtie B} g$ says that for every $x \in A$ if the computation $fx$ terminates, then $gx$ also terminates and both computations yield the same result.

Definition  Let $A$ be a class. A function $f \in (A \rightarrow A)$ is called $A$-monotonic, if

$$
\forall g \in A. \forall h \in A. g \sqsubseteq_A h \rightarrow f g \sqsubseteq_A f h.
$$
The Least Fixed Point Operator

Lemma  There exist closed terms $l$ and $h$ such that LFP proves:

1. $l\ g\ \downarrow$,
2. $l\ g\ x\ \downarrow\ \iff\ \exists n.\ h\ g\ n\ x\ \downarrow$,
3. $l\ g\ x = z\ \implies\ \exists n.\ h\ g\ n\ x = z$.

We have to show later that we can replace the term $g\ (l\ g)$ by a “finite approximation” $g\ (h\ g\ n)$.

\[
\begin{align*}
  hgn & \simeq \begin{cases} 
  b & \text{if } n = 0, \\
  g(hg(p_0 n)) & \text{otherwise}.
  \end{cases} \\
  qgxn & \simeq \begin{cases} 
  0 & \text{if } hg(p_0 n)x = p_1 n, \\
  \text{not}_N & \text{otherwise}.
  \end{cases}
\end{align*}
\]

\[
l := \lambda g\lambda x.\ p_1(p_0(\mu(\lambda y. c^3(q, g, x, p_0(y), p_1(y))))).
\]
**Lemma**  If $g \in ((A \rightsquigarrow B) \rightarrow (A \rightsquigarrow B))$ is $A \rightsquigarrow B$-monotonic, then the following claims hold in LFP:

1. $\forall n. h \, gn \in A \rightsquigarrow B$,
2. $\forall n. h \, gn \unrhd_{A \rightsquigarrow B} h\, g\,(n + 1)$,
3. $l\, g \in A \rightsquigarrow B$,
4. $\forall n. h \, gn \unrhd_{A \rightsquigarrow B} l\, g$,
5. $l\, g \unrhd_{A \rightsquigarrow B} g\,(l\, g)$,
6. $\forall l. \exists n. \forall x \in A. x \leq l \rightarrow l\, gx \subseteq h\, gn\, x$,
7. $\forall x \in A. \exists n. g\,(l\, g)\, x \subseteq g\,(h\, gn)\, x$.

**Theorem**  Let $g \in ((A \rightsquigarrow B) \rightarrow (A \rightsquigarrow B))$ be $A \rightsquigarrow B$-monotonic

1. $l\, g \equiv_{A \rightsquigarrow B} g\,(l\, g)$,
2. $f \in A \rightsquigarrow B \land f \equiv_{A \rightsquigarrow B} g\, f \rightarrow l\, g \unrhd_{A \rightsquigarrow B} f$. 
Example 1

Consider the following JAVA like method:

```java
A m (B x) {
    return m(x);
}
```

The semantics of the method \( m \) is given as the least fixed point of the functional \( \lambda f. \lambda x. f x \) which is the everywhere undefined function.

\[
\text{BON} \nVDash \neg \text{rec} (\lambda f. \lambda x. f x) s \downarrow
\]

Let \( V \) be the universal class \( x = x \) and \( \emptyset \) the empty class \( x \neq x \). Then the functional \( \lambda f. \lambda x. f x \) is an element of \( (V \bowtie \emptyset) \rightarrow (V \bowtie \emptyset) \) and is of course \( V \bowtie \emptyset \)-monotonic.

\[
\text{LFP} \vdash I (\lambda f. \lambda x. f x) \in (V \bowtie \emptyset)
\]

\[
\text{LFP} \vdash \forall y. \neg I (\lambda f. \lambda x. f x) y \downarrow
\]
Example 2

\[
f_1 x \simeq \begin{cases} 
1 & \text{if } x = 1, \\
\text{not}_N & \text{otherwise}
\end{cases}
\]

\[
f_2 x \simeq \begin{cases} 
\text{not}_N & \text{if } x = 1, \\
1 & \text{otherwise.}
\end{cases}
\]

\[
g x \simeq \begin{cases} 
f_1 & \text{if } x = f_1, \\
f_2 & \text{otherwise.}
\end{cases}
\]

We have \( g \in ((V \curvearrowright V) \to (V \curvearrowright V)) \).

If \( f \) is a fixed point of \( g \) then we have either \( f = f_1 \) or \( \forall x. f \ x \simeq f_2 \ x \).

However, \( g \) does not have a least fixed point in the sense of \( \sqsubseteq_{(V \curvearrowright V)} \), since it is not \((V \curvearrowright V)\)-monotonic:

\( f_1 \sqsubseteq_{(V \curvearrowright V)} \lambda x.1, \text{ but } \neg (g \ f_1 \sqsubseteq_{(V \curvearrowright V)} g \ (\lambda x.1)) \)