Proof interpretations: a modern perspective

Lecture 2 - Higher order computation

Anupam Das & Thomas Powell

University of Copenhagen & Technische Universität Darmstadt

North American Summer School on Logic, Language, and Information

Carnegie Mellon University

26 June 2018

These slides are available at `http://www.anupamdas.com/nasslli18`.

Some references

Useful references for this lecture

- Avigad, J. and Feferman, S. (1998). Gödel's functional ("Dialectica") interpretation.
  In Buss, S. R., editor, *Handbook of Proof Theory*, volume 137, pages 337–405. Elsevier.
  http://www.andrew.cmu.edu/user/avigad/Papers/dialect.pdf

- Boolos, G. S. and Jeffrey, R. C. (1989). *Computability and Logic: 3rd Ed.*
  Cambridge University Press, New York, NY, USA

- Troelstra, A. S. and Schwichtenberg, H. (1996). *Basic Proof Theory*.
  Cambridge University Press, New York, NY, USA

- Gödel, K. (1958). Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes.
  Dialectica, 12:280–287

- Kohlenbach, U. (2008). *Applied Proof Theory - Proof Interpretations and their Use in Mathematics*.
  Springer Monographs in Mathematics. Springer

Outline

1. Functions and computability
2. Primitive recursion
3. Higher types: first steps
4. System T
5. Primitive recursion and beyond in System T
6. Some metalogical properties of system T
7. Approaches via well-founded orders
8. References

Functions on $\mathbb{N}$

A **function** $f : \mathbb{N} \to \mathbb{N}$ takes a number as input and returns a number as output.
E.g.:

$$\begin{aligned} \mathsf{0}(n) &= 0 \\ \mathsf{id}(n) &= n \\ \mathsf{s}(n) &= n+1 \\ f_g(n) &= g(g(n)) \end{aligned} \qquad \mathsf{par}(n) = \begin{cases} 0 & n \text{ is even} \\ 1 & \text{otherwise} \end{cases}$$

$$f_{\varphi(x)}(n) = \begin{cases} 1 & \text{if } \varphi(n) \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

Functions on $\mathbb{N}$

A **function** $f : \mathbb{N} \to \mathbb{N}$ takes a number as input and returns a number as output.
E.g.:

$$\begin{aligned} \mathsf{0}(n) &= 0 \\ \mathsf{id}(n) &= n \\ \mathsf{s}(n) &= n+1 \\ f_g(n) &= g(g(n)) \end{aligned} \qquad \begin{aligned} \mathsf{par}(n) &= \begin{cases} 0 & n \text{ is even} \\ 1 & \text{otherwise} \end{cases} \\ f_{\varphi(x)}(n) &= \begin{cases} 1 & \text{if } \varphi(n) \text{ is true} \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

We can also have functions with many inputs, of 'type' $\mathbb{N} \times \cdots \times \mathbb{N} \to \mathbb{N}$.
E.g.:

$$\begin{aligned} \mathsf{add}(m,n) &= m+n \\ \mathsf{mult}(m,n) &= m^n \\ f(m,n,p,q) &= mn+pq \end{aligned}$$

$f(m,n) = \mathrm{g}(n)$
where g is the m^{th} program.

Computability of functions

We may distinguish an important class of functions:

Definition (informal)

A function $f : \mathbb{N}^k \to \mathbb{N}$ is **computable** if there is a program/algorithm that, when fed inputs $n_1, \ldots, n_k \in \mathbb{N}$, eventually terminates and returns $f(n)$.

Computability of functions

We may distinguish an important class of functions:

Definition (informal)

A function $f : \mathbb{N}^k \to \mathbb{N}$ is **computable** if there is a program/algorithm that, when fed inputs $n_1, \ldots, n_k \in \mathbb{N}$, eventually terminates and returns $f(n)$.

Question: What is a program or algorithm?

Computability of functions

We may distinguish an important class of functions:

Definition (informal)

A function $f : \mathbb{N}^k \to \mathbb{N}$ is **computable** if there is a program/algorithm that, when fed inputs $n_1, \ldots, n_k \in \mathbb{N}$, eventually terminates and returns $f(n)$.

Question: What is a program or algorithm?

In this course we will not concern ourselves with formal definitions, but point out:

A. Church's thesis, 1936

It does not matter which programming language we use! They are all equivalent.

Example: computing sums

The addition function

add : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is given by $(m, n) \mapsto m + n$.

Example: computing sums

The addition function

add : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is given by $(m, n) \mapsto m + n$.

A basic program

1: **inputs:** $m, n \in \mathbb{N}$.
2: **if** $m = 0$ **then**
3: **return** n
4: **else**
5: **let** $m := m - 1$
6: **let** $n := n + 1$
7: **end if**
8: **goto** 2.

Example: computing sums

The addition function

add : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is given by $(m, n) \mapsto m + n$.

A basic program

1: **inputs:** $m, n \in \mathbb{N}$.
2: **if** $m = 0$ **then**
3: **return** n
4: **else**
5: **let** $m := m - 1$
6: **let** $n := n + 1$
7: **end if**
8: **goto** 2.

An example 'run'

If m, n are initialised to 3, 2 respectively, this is how their values change as the program runs,

$$\begin{aligned} (3,2) &\to_{5,6} (2,3) \\ &\to_{5,6} (1,4) \\ &\to_{5,6} (0,5) \end{aligned}$$

at which point the program returns 5, by line 3.

Example: computing sums

The addition function

add : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is given by $(m, n) \mapsto m + n$.

A basic program

1: **inputs:** $m, n \in \mathbb{N}$.
2: **if** $m = 0$ **then**
3: **return** n
4: **else**
5: **let** $m := m - 1$
6: **let** $n := n + 1$
7: **end if**
8: **goto** 2.

An example 'run'

If m, n are initialised to 3, 2 respectively, this is how their values change as the program runs,

$$\begin{aligned} (3,2) &\to_{5,6} (2,3) \\ &\to_{5,6} (1,4) \\ &\to_{5,6} (0,5) \end{aligned}$$

at which point the program returns 5, by line 3.

Question: Is this the most efficient algorithm for addition?

Example: computing sums

The addition function

add : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is given by $(m, n) \mapsto m + n$.

A basic program

1: **inputs:** $m, n \in \mathbb{N}$.
2: **if** $m = 0$ **then**
3: **return** n
4: **else**
5: **let** $m := m - 1$
6: **let** $n := n + 1$
7: **end if**
8: **goto** 2.

An example 'run'

If m, n are initialised to 3, 2 respectively, this is how their values change as the program runs,

$$\begin{aligned} (3,2) &\to_{5,6} (2,3) \\ &\to_{5,6} (1,4) \\ &\to_{5,6} (0,5) \end{aligned}$$

at which point the program returns 5, by line 3.

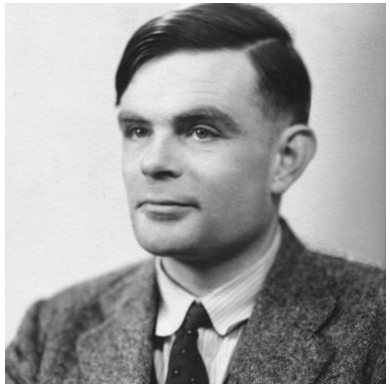
Question: Is this the most efficient algorithm for addition?

NB: Commands like **goto** or **while**, capable of producing infinite loops, are necessary for a programming language to be complete.

The futility of debugging

Theorem (A. Turing, 1936)

There is no procedure deciding whether a given program terminates on a given input.



The futility of debugging

Theorem (A. Turing, 1936)

There is no procedure deciding whether a given program terminates on a given input.

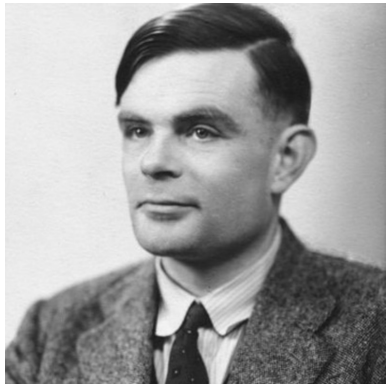
Informal proof.

Suppose we could. Define a program D on other programs P as follows:

$$D(P) \quad := \quad \begin{cases} \texttt{loop} & \text{if } P(P) \text{ terminates} \\ \texttt{0} & \text{otherwise} \end{cases}$$

What happens when I run $D(D)$? □

The futility of debugging



Theorem (A. Turing, 1936)

There is no procedure deciding whether a given program terminates on a given input.

Informal proof.

Suppose we could. Define a program D on other programs P as follows:

$$D(P) \quad := \quad \begin{cases} \texttt{loop} & \text{if } P(P) \text{ terminates} \\ \texttt{0} & \text{otherwise} \end{cases}$$

What happens when I run $D(D)$? □

Our motivating question: Which functions/programs can we *prove* are well-defined?

In this course we focus on the case of what we can prove in theories of arithmetic, but the techniques are general.

Outline

1. Functions and computability
2. Primitive recursion
3. Higher types: first steps
4. System T
5. Primitive recursion and beyond in System T
6. Some metalogical properties of system T
7. Approaches via well-founded orders
8. References

Primitive recursion: the logician's playground

In the 1920s-30s, logicians such as Skolem, Péter and Ackermann proposed the *primitive recursive* class of computable functions.

Primitive recursion: the logician's playground

In the 1920s-30s, logicians such as Skolem, Péter and Ackermann proposed the *primitive recursive* class of computable functions.

Definition

We say that a function $f(x, \vec{x})$ on $\mathbb{N}$ is defined by **primitive recursion** from functions $g(\vec{x})$, $h(x, \vec{x}, y)$ if:

$$\begin{aligned} f(0, \vec{x}) &= g(\vec{x}) \\ f(\mathsf{s}(x), \vec{x}) &= h(x, \vec{x}, f(x, \vec{x})) \end{aligned} \tag{1}$$

Primitive recursion: the logician's playground

In the 1920s-30s, logicians such as Skolem, Péter and Ackermann proposed the *primitive recursive* class of computable functions.

Definition

We say that a function $f(x, \vec{x})$ on $\mathbb{N}$ is defined by **primitive recursion** from functions $g(\vec{x})$, $h(x, \vec{x}, y)$ if:

$$\begin{aligned} f(0, \vec{x}) &= g(\vec{x}) \\ f(\mathsf{s}(x), \vec{x}) &= h(x, \vec{x}, f(x, \vec{x})) \end{aligned} \tag{1}$$

The *primitive recursive functions* are the least class of functions on $\mathbb{N}$ containing,

$$\begin{aligned} 0(\cdot) &:= 0 \\ \mathsf{s}(x) &:= x + 1 \\ P_i^k(x_0, \ldots, x_{k-1}) &= x_i \qquad \text{for all } i, k \in \mathbb{N} \text{ with } i < k \end{aligned}$$

and closed under composition and primitive recursion.

NB: $f(x, \vec{x})$ is defined by **composition** from functions $g(x, \vec{x})$, $h(x, \vec{x}, y)$ if $f(x, \vec{x}) = h(x, \vec{x}, g(x, \vec{x}))$.

Some examples

$$
\begin{aligned}
\mathsf{p}(0) &= 0 \\
\mathsf{p}(\mathsf{s}x) &= x \\
x \dot{-} 0 &= x \\
x \dot{-} (\mathsf{s}y) &= (x \dot{-} y) \dot{-} 1 \\
\mathsf{add}(0, y) &= y \\
\mathsf{add}(\mathsf{s}x, y) &= \mathsf{s}(\mathsf{add}(x, y))
\end{aligned}
$$

Some examples

$$
\begin{aligned}
\mathsf{p}(0) &= 0 \\
\mathsf{p}(\mathsf{s}x) &= x \\
x \dot{-} 0 &= x \\
x \dot{-} (\mathsf{s}y) &= (x \dot{-} y) \dot{-} 1 \\
\mathsf{add}(0, y) &= y \\
\mathsf{add}(\mathsf{s}x, y) &= \mathsf{s}(\mathsf{add}(x, y))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{zero}(0) &= 1 \\
\mathsf{zero}(\mathsf{s}x) &= 0 \\
\mathsf{par}(0) &= 0 \\
\mathsf{par}(\mathsf{s}x) &= 1 \dot{-} \mathsf{par}(x)
\end{aligned}
$$

$$
\begin{aligned}
& \texttt{if } x \texttt{ is odd then } y \texttt{ else } z \\
= \; & y \cdot \mathsf{par}(x) + z \cdot (1 \dot{-} \mathsf{par}(x))
\end{aligned}
$$

Some examples

$$
\begin{aligned}
\mathsf{p}(0) &= 0 \\
\mathsf{p}(\mathsf{s}x) &= x \\
x \dot{-} 0 &= x \\
x \dot{-} (\mathsf{s}y) &= (x \dot{-} y) \dot{-} 1 \\
\mathsf{add}(0, y) &= y \\
\mathsf{add}(\mathsf{s}x, y) &= \mathsf{s}(\mathsf{add}(x, y))
\end{aligned}
\qquad
\begin{aligned}
\mathsf{zero}(0) &= 1 \\
\mathsf{zero}(\mathsf{s}x) &= 0 \\
\mathsf{par}(0) &= 0 \\
\mathsf{par}(\mathsf{s}x) &= 1 \dot{-} \mathsf{par}(x) \\
&\texttt{if } x \texttt{ is odd then } y \texttt{ else } z \\
&= y \cdot \mathsf{par}(x) + z \cdot (1 \dot{-} \mathsf{par}(x))
\end{aligned}
$$

NB: Almost all common computer programs are primitive recursive!

Some examples

$$
\begin{aligned}
\mathsf{p}(0) &= 0 \\
\mathsf{p}(\mathsf{s}x) &= x \\
x \dot{-} 0 &= x \\
x \dot{-} (\mathsf{s}y) &= (x \dot{-} y) \dot{-} 1 \\
\mathsf{add}(0, y) &= y \\
\mathsf{add}(\mathsf{s}x, y) &= \mathsf{s}(\mathsf{add}(x, y))
\end{aligned}
\qquad
\begin{aligned}
\mathsf{zero}(0) &= 1 \\
\mathsf{zero}(\mathsf{s}x) &= 0 \\
\mathsf{par}(0) &= 0 \\
\mathsf{par}(\mathsf{s}x) &= 1 \dot{-} \mathsf{par}(x) \\
&\texttt{if } x \texttt{ is odd then } y \texttt{ else } z \\
&= y \cdot \mathsf{par}(x) + z \cdot (1 \dot{-} \mathsf{par}(x))
\end{aligned}
$$

NB: Almost all common computer programs are primitive recursive!

Exercise: Give primitive recursive programs for $x \times y$ and x^y. Can you construct even 'bigger' functions?

Jumping ranks: outgrowing primitive recursion

Definition (Ackermann-Péter function)

$$\begin{aligned} A(0,n) &= n+1 \\ A(m+1,0) &= A(m,1) \\ A(m+1,n+1) &= A(m,A(m+1,n)) \end{aligned}$$

Jumping ranks: outgrowing primitive recursion

Definition (Ackermann-Péter function)

$$\begin{aligned} A(0,n) &= n+1 \\ A(m+1,0) &= A(m,1) \\ A(m+1,n+1) &= A(m,A(m+1,n)) \end{aligned}$$

Convince yourselves that:

1. A is, indeed, a function $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$. I.e. it is defined everywhere.
2. A is, in fact, computable. I.e. there is an algorithm computing it.

Jumping ranks: outgrowing primitive recursion

Definition (Ackermann-Péter function)

$$\begin{aligned} A(0,n) &= n+1 \\ A(m+1,0) &= A(m,1) \\ A(m+1,n+1) &= A(m,A(m+1,n)) \end{aligned}$$

Convince yourselves that:

1. A is, indeed, a function $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$. I.e. it is defined everywhere.
2. A is, in fact, computable. I.e. there is an algorithm computing it.

Theorem (W. Ackermann, 1928)

$A(m,n)$ is not primitive recursive.

Intuition: $A(m,n)$ grows faster than any primitive recursive function.

Unbounded search: computability in general

As we already mentioned, commands that allow infinite loops or unbounded search are required in order to express all computable functions.

Unbounded search: computability in general

As we already mentioned, commands that allow infinite loops or unbounded search are required in order to express all computable functions.

In fact, we may add a natural such feature to primitive recursive programs:

$$\mu x.(f(x,\vec{x}) = 0) \;:=\; \text{the least } x \text{ s.t. } f(x,\vec{x}) = 0, \text{ if such } x \text{ exists} \tag{2}$$

NB: Does not always return an output! An unbounded search might not terminate.

Unbounded search: computability in general

As we already mentioned, commands that allow infinite loops or unbounded search are required in order to express all computable functions.

In fact, we may add a natural such feature to primitive recursive programs:

$$\mu x.(f(x,\vec{x}) = 0) \;:=\; \text{the least } x \text{ s.t. } f(x,\vec{x}) = 0, \text{ if such } x \text{ exists} \tag{2}$$

NB: Does not always return an output! An unbounded search might not terminate.

It turns out that this simple extension allows us to express all computable functions:

Proposition

The computable functions are the least class of functions containing containing the primitive recursive functions and closed under minimisation, (2).

Unbounded search: computability in general

As we already mentioned, commands that allow infinite loops or unbounded search are required in order to express all computable functions.

In fact, we may add a natural such feature to primitive recursive programs:

$$\mu x.(f(x,\vec{x}) = 0) \;:=\; \text{the least } x \text{ s.t. } f(x,\vec{x}) = 0, \text{ if such } x \text{ exists} \tag{2}$$

NB: Does not always return an output! An unbounded search might not terminate.

It turns out that this simple extension allows us to express all computable functions:

Proposition

The computable functions are the least class of functions containing containing the primitive recursive functions and closed under minimisation, (2).

Corollary

(2) *may express functions not provably well-defined in arithmetic.*

Summing up: motivation

Motivating question

Is there a programming language that expresses precisely the functions that are provably well-defined in, say, arithmetic?

Summing up: motivation

Motivating question

Is there a programming language that expresses precisely the functions that are provably well-defined in, say, arithmetic?

We have seen that:

- The primitive recursive functions are not powerful enough, due to the inexpressibility of the Ackermann-Péter function.
- The computable functions exceed arithmetic.

We would like something in between, that retains the structured nature of primitive recursion and enjoys guaranteed termination, rather than adding the full weight of unbounded search.

Summing up: motivation

Motivating question

Is there a programming language that expresses precisely the functions that are provably well-defined in, say, arithmetic?

We have seen that:

- The primitive recursive functions are not powerful enough, due to the inexpressibility of the Ackermann-Péter function.
- The computable functions exceed arithmetic.

We would like something in between, that retains the structured nature of primitive recursion and enjoys guaranteed termination, rather than adding the full weight of unbounded search.

One way to do this is to allow primitive recursion over *functionals*…

Outline

1. Functions and computability
2. Primitive recursion
3. Higher types: first steps
4. System T
5. Primitive recursion and beyond in System T
6. Some metalogical properties of system T
7. Approaches via well-founded orders
8. References

Functionals on $\mathbb{N}$

A **functional** may take a function itself as an input.
E.g.:

$$\begin{array}{rcll} F(f) & = & 0 & (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \\ F(f) & = & f(0) & (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \\ F(g,n) & = & g(g(n)) & (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \to \mathbb{N} \end{array}$$

Functionals on $\mathbb{N}$

A **functional** may take a function itself as an input.
E.g.:

$$\begin{array}{rcll} F(f) & = & 0 & (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \\ F(f) & = & f(0) & (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \\ F(g,n) & = & g(g(n)) & (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \to \mathbb{N} \end{array}$$

Functionals may also return functions.
E.g.:

$$\begin{array}{rcll} F(f) & = & f & (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}) \\ F(f) & = & (n \mapsto f(n) + f(n+1)) & (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}) \\ F(f,m) & = & (n \mapsto f(n), m-1) & (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \end{array}$$

Functionals on $\mathbb{N}$

A **functional** may take a function itself as an input.
E.g.:

$$\begin{array}{rcll} F(f) & = & 0 & (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \\ F(f) & = & f(0) & (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \\ F(g,n) & = & g(g(n)) & (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \to \mathbb{N} \end{array}$$

Functionals may also return functions.
E.g.:

$$\begin{array}{rcll} F(f) & = & f & (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}) \\ F(f) & = & (n \mapsto f(n) + f(n+1)) & (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}) \\ F(f,m) & = & (n \mapsto f(n), m-1) & (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \end{array}$$

This is starting to get messy!

A useful notation: We write $\lambda n.f(n)$ for the function $n \mapsto f(n)$.

E.g.: add is the function $\lambda m.\lambda n.m + n$.

Computable functionals, briefly

What does it mean to be a computable functional?

Computable functionals, briefly

What does it mean to be a computable functional?

Definition (informal)

We say that a function f is computable in g if there is an algorithm computing f that is allowed to 'freely calculate' values of g.

Intuition: we treat input functions as black boxes, called **oracles** which may be freely *queried* by an algorithm.

Computable functionals, briefly

What does it mean to be a computable functional?

Definition (informal)

We say that a function f is computable in g if there is an algorithm computing f that is allowed to 'freely calculate' values of g.

Intuition: we treat input functions as black boxes, called **oracles** which may be freely *queried* by an algorithm.

E.g.: The function $F = \lambda g.\lambda n.g(g(n))$ is computable in g.

Outline

1. Functions and computability
2. Primitive recursion
3. Higher types: first steps
4. System T
5. Primitive recursion and beyond in System T
6. Some metalogical properties of system T
7. Approaches via well-founded orders
8. References

Background

Background

Gödel introduced **system** T in order to express all the functions that are well-defined in arithmetic.

System T *trades off* logical complexity for computational complexity:

- it is a quantifier-free theory;
- its terms form a rich programming language.

Background

Gödel introduced **system** T in order to express all the functions that are well-defined in arithmetic.

System T *trades off* logical complexity for computational complexity:

- it is a quantifier-free theory;
- its terms form a rich programming language.

The terms/programs of system T combine two fundamental ideas we have seen:

- Primitive recursion.
- Higher-type functions.

It turns out that this is precisely what we need for arithmetic!

Background

Gödel introduced **system** T in order to express all the functions that are well-defined in arithmetic.

System T *trades off* logical complexity for computational complexity:

- it is a quantifier-free theory;
- its terms form a rich programming language.

The terms/programs of system T combine two fundamental ideas we have seen:

- Primitive recursion.
- Higher-type functions.

It turns out that this is precisely what we need for arithmetic!

We will now:

- Define system T formally;
- Show how its terms/programs strictly extend primitive recursion; and
- See how it may be viewed as a programming language.

Types

We have already informally discussed the notion of 'type'. Let us now be formal.

Definition (Types)

Types, written ρ, σ, etc., are defined as follows:

- N is a type.
- If ρ and σ are types, then so is $(\rho \to \sigma)$.

Intuition: We imagine N as representing the set of natural numbers, $N \to N$ as representing functions, $(N \to N) \to N$ functionals and so on.

Types

We have already informally discussed the notion of 'type'. Let us now be formal.

Definition (Types)

Types, written ρ, σ, etc., are defined as follows:

- N is a type.
- If ρ and σ are types, then so is $(\rho \to \sigma)$.

Intuition: We imagine N as representing the set of natural numbers, $N \to N$ as representing functions, $(N \to N) \to N$ functionals and so on. Formally:

Definition (Levels)

We define the **level** or **order** of a type σ, written $\text{level}(\sigma)$, as follows:

- $\text{level}(N) := 0$.
- $\text{level}(\rho \to \tau) = \max\{\text{level}(\rho) + 1, \text{level}(\tau)\}$.

Terms

As we mentioned, system T can be viewed as a higher-order version of the primitive recursive functions.

Terms

As we mentioned, system T can be viewed as a higher-order version of the primitive recursive functions. Defining its programs, however, is a little more complicated:

Definition (Typed terms)

For all types ρ, σ, τ, the language of system T includes:

- Infinitely many variables x^ρ, y^ρ, z^ρ, etc.
- A constant $\mathsf{0}$ of type N.
- A constant s of type $N \to N$.
- A constant $K_{\rho,\sigma}$ of type $\rho \to (\sigma \to \rho)$.
- A constant $S_{\rho,\sigma,\tau}$ of type $(\rho \to (\sigma \to \tau)) \to ((\rho \to \sigma) \to (\rho \to \tau))$.
- A constant R_ρ of type $\rho \to ((N \to (\rho \to \rho)) \to (N \to \rho))$.

Terms

As we mentioned, system T can be viewed as a higher-order version of the primitive recursive functions. Defining its programs, however, is a little more complicated:

Definition (Typed terms)

For all types ρ, σ, τ, the language of system T includes:

- Infinitely many variables x^ρ, y^ρ, z^ρ, etc.
- A constant $\mathsf{0}$ of type N.
- A constant s of type $N \to N$.
- A constant $K_{\rho,\sigma}$ of type $\rho \to (\sigma \to \rho)$.
- A constant $S_{\rho,\sigma,\tau}$ of type $(\rho \to (\sigma \to \tau)) \to ((\rho \to \sigma) \to (\rho \to \tau))$.
- A constant R_ρ of type $\rho \to ((N \to (\rho \to \rho)) \to (N \to \rho))$.

The (typed) **terms** of system T, written s, t, etc., are defined as follows:

- All variables and constants are terms of their indicated types.
- If t is a term of type $\rho \to \sigma$ and s is a term of type ρ then $t(s)$ is a term of type σ.

Some examples and conventions

The terms $\mathsf{0}, \mathsf{s}(\mathsf{0}), \mathsf{ss}(\mathsf{0}), \ldots, \mathsf{s}^k(\mathsf{0})$ are called numerals. They represent the natural numbers and have type N, so have level $\mathsf{0}$.

Some examples and conventions

The terms $\mathsf{0}, \mathsf{s}(\mathsf{0}), \mathsf{ss}(\mathsf{0}), \ldots, \mathsf{s}^k(\mathsf{0})$ are called numerals. They represent the natural numbers and have type N, so have level 0.

Other examples:

- The term $K_{N,N}(\mathsf{ss}(\mathsf{0}))$ has type $N \to N$, so has level 1.
- The term $R_N(x^N)$ has type $(N \to (N \to N)) \to (N \to N)$, so has level 2.

However, it is not clear what these terms *mean*, but very soon we will give some axioms governing them.

Some examples and conventions

The terms $\mathsf{0}, \mathsf{s}(\mathsf{0}), \mathsf{ss}(\mathsf{0}), \ldots, \mathsf{s}^k(\mathsf{0})$ are called numerals. They represent the natural numbers and have type N, so have level 0.

Other examples:

- The term $K_{N,N}(\mathsf{ss}(\mathsf{0}))$ has type $N \to N$, so has level 1.
- The term $R_N(x^N)$ has type $(N \to (N \to N)) \to (N \to N)$, so has level 2.

However, it is not clear what these terms *mean*, but very soon we will give some axioms governing them.

Notation: Writing out types can be long and cumbersome. Henceforth we may omit parantheses in long expressions under the following convention:

$$\sigma_1 \to \sigma_2 \to \cdots \sigma_k \to \tau \quad := \quad \sigma_1 \to (\sigma_2 \to (\cdots (\sigma_k \to \tau) \cdots))$$

I.e., we assume parantheses are *associated to the right*.
E.g., the type of the term $R_N(x^N)$ is simply written $(N \to N \to N) \to N \to N$

We often mentally construe a type $N \to \cdots \to N \to N$ rather as $N \times \cdots \times N \to N$.
E.g., we may write $K_{\rho,\sigma}(x, y)$ instead of $(K_{\rho,\sigma}(x))(y)$.

Giving meaning to the combinators

The constants $K_{\rho,\sigma}$ and $S_{\rho,\sigma,\tau}$ are called the **combinators**, and allow us to build functions from terms.

Giving meaning to the combinators

The constants $K_{\rho,\sigma}$ and $S_{\rho,\sigma,\tau}$ are called the **combinators**, and allow us to build functions from terms.

System T includes the following axioms for them,

$$\begin{aligned} K_{\rho,\sigma}(u,v) &= u \\ S_{\rho,\sigma,\tau}(x,y,z) &= x(z)(y(z)) \end{aligned}$$

for all types ρ, σ, τ where:

- u has type ρ and v has type σ,
- x has type $\rho \to \sigma \to \tau$, y has type $\rho \to \sigma$ and z has type τ.

Giving meaning to the combinators

The constants $K_{\rho,\sigma}$ and $S_{\rho,\sigma,\tau}$ are called the **combinators**, and allow us to build functions from terms.

System T includes the following axioms for them,

$$\begin{aligned} K_{\rho,\sigma}(u,v) &= u \\ S_{\rho,\sigma,\tau}(x,y,z) &= x(z)(y(z)) \end{aligned}$$

for all types ρ, σ, τ where:

- u has type ρ and v has type σ,
- x has type $\rho \to \sigma \to \tau$, y has type $\rho \to \sigma$ and z has type τ.

NB: Notice the similarity between the combinators $K_{\rho,\sigma}$ and $S_{\rho,\sigma,\tau}$ and the projection functions and composition operation for primitive recursive functions.

Combinatory completeness

Proposition (Combinatory completeness)

For each term t of type σ and variable x of type ρ, there exists a term $\lambda x.t$ of type $\rho \to \sigma$, whose free variables are those of t without x, and which satisfies provably in T*:*

$$(\lambda x.t)(s) = t[s/x]$$

Combinatory completeness

Proposition (Combinatory completeness)

For each term t of type σ and variable x of type ρ, there exists a term $\lambda x.t$ of type $\rho \to \sigma$, whose free variables are those of t without x, and which satisfies provably in T*:*

$$(\lambda x.t)(s) = t[s/x]$$

NB: This result is a refinement of the completeness of propositional logic.

Combinatory completeness

Proposition (Combinatory completeness)

For each term t of type σ and variable x of type ρ, there exists a term $\lambda x.t$ of type $\rho \to \sigma$, whose free variables are those of t without x, and which satisfies provably in T*:*

$$(\lambda x.t)(s) = t[s/x]$$

NB: This result is a refinement of the completeness of propositional logic.

Example

For any variable x^ρ we may define a term $\lambda x.x$ as $S(K,K)$. Notice that:

$$\begin{array}{lcll} (S(K,K))(x) & = & S(K,K,x) & \\ & = & K(x)(K(x)) & \text{by the } S \text{ axioms} \\ & = & K(x,K(x)) & \\ & = & x & \text{by the } K \text{ axioms} \end{array}$$

Combinatory completeness

Proposition (Combinatory completeness)

For each term t of type σ and variable x of type ρ, there exists a term $\lambda x.t$ of type $\rho \to \sigma$, whose free variables are those of t without x, and which satisfies provably in T*:*

$$(\lambda x.t)(s) = t[s/x]$$

NB: This result is a refinement of the completeness of propositional logic.

Example

For any variable x^ρ we may define a term $\lambda x.x$ as $S(K,K)$. Notice that:

$$\begin{array}{lcll} (S(K,K))(x) & = & S(K,K,x) & \\ & = & K(x)(K(x)) & \text{by the } S \text{ axioms} \\ & = & K(x,K(x)) & \\ & = & x & \text{by the } K \text{ axioms} \end{array}$$

Exercise: Work out the correct types for the combinators above.

Recursion in system T

The constants R_ρ are called **recursors** and allow us to construct functionals via recursion.

Recursion in system T

The constants R_ρ are called **recursors** and allow us to construct functionals via recursion.

System T includes the following axioms for them,

$$\begin{aligned} R_\rho(g, h, 0) &= g \\ R_\rho(g, h, \mathsf{s}(n)) &= h(n, R_\rho(g, h, n)) \end{aligned}$$

for all types ρ where g has type ρ, h has type $N \to \rho \to \rho$ and n is a numeral.

Recursion in system T

The constants R_ρ are called **recursors** and allow us to construct functionals via recursion.

System T includes the following axioms for them,

$$\begin{array}{rcl} R_\rho(g,h,0) & = & g \\ R_\rho(g,h,s(n)) & = & h(n,R_\rho(g,h,n)) \end{array}$$

for all types ρ where g has type ρ, h has type $N \to \rho \to \rho$ and n is a numeral.

NB: Again, notice the similarity between the recursors R_ρ and the operation of primitive recursion. Writing $f(n) = R_\rho(g,h,n)$, we have:

$$\begin{array}{rcl} f(0) & = & g \\ f(s(n)) & = & h(n,f(n)) \end{array}$$

Building a logical theory

Formulae

For terms s, t of the same type, $s = t$ is a formula.
If φ, ψ are formulae then $\varphi \vee \psi$, $\varphi \wedge \psi$ and $\varphi \to \psi$ are formulae.

Building a logical theory

Formulae

For terms s, t of the same type, $s = t$ is a formula.
If φ, ψ are formulae then $\varphi \vee \psi$, $\varphi \wedge \psi$ and $\varphi \to \psi$ are formulae.

Equality axioms

- $x^\rho = x^\rho$.
- $x^\rho = y^\rho \to \varphi[x^\rho/z^\rho] \to \varphi[y^\rho/z^\rho]$.

Substitution rule

$$\frac{\varphi}{\varphi[t^\rho/x^\rho]}$$

Propositional logic

All theorems of classical propositional logic are axioms of system T.

Building a logical theory

Formulae

For terms s, t of the same type, $s = t$ is a formula.
If φ, ψ are formulae then $\varphi \vee \psi$, $\varphi \wedge \psi$ and $\varphi \to \psi$ are formulae.

Equality axioms

- $x^\rho = x^\rho$.
- $x^\rho = y^\rho \to \varphi[x^\rho/z^\rho] \to \varphi[y^\rho/z^\rho]$.

Substitution rule

$$\frac{\varphi}{\varphi[t^\rho/x^\rho]}$$

Propositional logic

All theorems of classical propositional logic are axioms of system T.

Induction rule

$$\frac{\varphi(0) \quad \varphi(x^N) \to \varphi(\mathsf{s}(x^N))}{\varphi(t^N)}$$

Outline

1. Functions and computability
2. Primitive recursion
3. Higher types: first steps
4. System T
5. Primitive recursion and beyond in System T
6. Some metalogical properties of system T
7. Approaches via well-founded orders
8. References

The primitive recursive functions in system T

Proposition

Every primitive recursive function is computed by a typed term of system T.

The primitive recursive functions in system T

Proposition

Every primitive recursive function is computed by a typed term of system T.

Proof idea.

For each primitive recursive function $f(x_1, \ldots, x_k)$ we construct a term of type $\overbrace{N \to \cdots \to N}^{k} \to N$ computing it, by induction on the program structure of f.

The primitive recursive functions in system T

Proposition

Every primitive recursive function is computed by a typed term of system T.

Proof idea.

For each primitive recursive function $f(x_1, \ldots, x_k)$ we construct a term of type $\overbrace{N \to \cdots \to N}^{k} \to N$ computing it, by induction on the program structure of f.

For the base cases, the initial primitive recursive functions $\mathsf{0}$, s, P_i^k have direct analogues in T by $\mathsf{0}$, s, K.

The primitive recursive functions in system T

Proposition

Every primitive recursive function is computed by a typed term of system T.

Proof idea.

For each primitive recursive function $f(x_1, \ldots, x_k)$ we construct a term of type $\overbrace{N \to \cdots \to N}^{k} \to N$ computing it, by induction on the program structure of f.

For the base cases, the initial primitive recursive functions $\mathsf{0}$, s, P_i^k have direct analogues in T by $\mathsf{0}$, s, K.

For the inductive steps, the composition operation for primitive recursive functions is simulated by the S combinator, while the primitive recursion operation is simulated by recursors of type N. □

The primitive recursive functions in system T

Proposition

Every primitive recursive function is computed by a typed term of system T.

Proof idea.

For each primitive recursive function $f(x_1, \ldots, x_k)$ we construct a term of type $\overbrace{N \to \cdots \to N}^{k} \to N$ computing it, by induction on the program structure of f.

For the base cases, the initial primitive recursive functions $\mathsf{0}$, s, P_i^k have direct analogues in T by $\mathsf{0}$, s, K.

For the inductive steps, the composition operation for primitive recursive functions is simulated by the S combinator, while the primitive recursion operation is simulated by recursors of type N. □

Exercise: Fill in the details of this proof.

Beyond primitive recursion via higher types

We only need to use recursors of type N, *i.e.* of level 0, when simulating primitive recursive functions. Higher levels afford us far greater expressivity.

Beyond primitive recursion via higher types

We only need to use recursors of type N, *i.e.* of level 0, when simulating primitive recursive functions. Higher levels afford us far greater expressivity. In particular:

Proposition

The Ackermann-Péter function is computed by a term in system T.

Beyond primitive recursion via higher types

We only need to use recursors of type N, *i.e.* of level 0, when simulating primitive recursive functions. Higher levels afford us far greater expressivity. In particular:

Proposition

The Ackermann-Péter function is computed by a term in system T.

Construction of the term, informal.

Let g be a variable of type $N \to N$. We may define a function $I(g)$ of type $N \to N$ by primitive recursion as follows:

$$\begin{aligned} I(g, 0) &= g(1) \\ I(g, \mathsf{s}(n)) &= g(I(g, n)) \end{aligned}$$

Intuition: $I(g, n) = g^{(n+1)}(1)$.

Beyond primitive recursion via higher types

We only need to use recursors of type N, *i.e.* of level 0, when simulating primitive recursive functions. Higher levels afford us far greater expressivity. In particular:

Proposition

The Ackermann-Péter function is computed by a term in system T.

Construction of the term, informal.

Let g be a variable of type $N \to N$. We may define a function $I(g)$ of type $N \to N$ by primitive recursion as follows:

$$\begin{aligned} I(g, 0) &= g(1) \\ I(g, \mathsf{s}(n)) &= g(I(g, n)) \end{aligned}$$

Intuition: $I(g, n) = g^{(n+1)}(1)$.

Now use recursion of type $N \to N$ to define A of type $N \to N \to N$ by

$$\begin{aligned} A(0) &= \lambda n.\mathsf{s}(n) \\ A(\mathsf{s}(m)) &= I(A(m)) \end{aligned}$$

□

Satisfying equational properties in T

Proposition

T *proves the defining equations* (4) *for term* A *we constructed. I.e. the following,*

$$\begin{aligned} A(\mathsf{0}, n) &= \mathsf{s}(n) \\ A(\mathsf{s}(m), \mathsf{0}) &= A(m, \mathsf{1}) \\ A(\mathsf{s}(m), \mathsf{s}(n)) &= A(m, A(\mathsf{s}(m), n)) \end{aligned}$$

are provable in T.

Satisfying equational properties in T

Proposition

T *proves the defining equations* (4) *for term* A *we constructed. I.e. the following,*

$$\begin{aligned} A(0, n) &= \mathsf{s}(n) \\ A(\mathsf{s}(m), 0) &= A(m, 1) \\ A(\mathsf{s}(m), \mathsf{s}(n)) &= A(m, A(\mathsf{s}(m), n)) \end{aligned}$$

are provable in T.

Proof.

By following definitions, we have:

$$\begin{aligned} & A(0, n) \\ = \; & A(0)(n) \\ = \; & (\lambda n.\mathsf{s}(n))n \\ = \; & \mathsf{s}(n) \end{aligned} \qquad \begin{aligned} & A(\mathsf{s}(m), 0) \\ = \; & A(\mathsf{s}(m))(0) \\ = \; & I(A(m)(0) \\ = \; & I(A(m), 0) \\ = \; & A(m, 1) \end{aligned} \qquad \begin{aligned} & A(\mathsf{s}m, \mathsf{s}n) \\ = \; & I(A(m), \mathsf{s}n) \\ = \; & A(m, I(A(m), n)) \\ = \; & A(m, A(\mathsf{s}m, n)) \end{aligned}$$

□

Break, and some exercises

Computability

1. Describe an algorithm for how you learned to add numbers at school.
2. Find primitive recursive functions $\langle m, n\rangle$, $\beta_0(n)$ and $\beta_1(n)$ s.t. $\beta_0\langle m, n\rangle = m$, $\beta_1\langle m, n\rangle = n$ and $\langle \beta_0(n), \beta_1(n)\rangle = n$.

Ackermann-Péter function

3. Calculate $A(2, 3)$ and $A(2, 4)$. What about $A(4, 2)$?
4. Show that, for any primitive recursive function $f(\vec{x})$ there is some $m \in \mathbb{N}$ s.t. $f(\vec{x}) < A(m, \max(\vec{x}))$. Hint: use induction on the *program structure* of f.
5. Conclude that the Ackermann-Péter function is not primitive recursive.

System T

6. Show, formally, that terms of system T with only recursors of level 0 suffice to compute the primitive recursive functions.
7. Write down *explicitly* the term that computes the Ackermann-Péter function.
8. Can you think of how to construct a computable function that is not computed by any term of system T?

Outline

1. Functions and computability
2. Primitive recursion
3. Higher types: first steps
4. System T
5. Primitive recursion and beyond in System T
6. Some metalogical properties of system T
7. Approaches via well-founded orders
8. References

Reduction and termination

The axioms we gave for the constants can be seen as a form of computation.

Orienting them left-right gives us the following notion of reduction:

$$\begin{array}{rcl@{\qquad\qquad}rcl} K_{\rho,\sigma}(s,t) & \triangleright & s & R_\rho(s,t,0) & \triangleright & g \\ S_{\rho,\sigma,\tau}(s,t,u) & \triangleright & s(u)(t(u)) & R_\rho(s,t,\mathsf{s}(n)) & \triangleright & t(n,R_\rho(s,t,n)) \end{array}$$

The RHSs are morally 'simpler' than the LHSs.

Reduction and termination

The axioms we gave for the constants can be seen as a form of computation.

Orienting them left-right gives us the following notion of reduction:

$$\begin{array}{rcl@{\qquad\qquad}rcl} K_{\rho,\sigma}(s,t) & \rhd & s & R_\rho(s,t,0) & \rhd & g \\ S_{\rho,\sigma,\tau}(s,t,u) & \rhd & s(u)(t(u)) & R_\rho(s,t,\mathsf{s}(n)) & \rhd & t(n,R_\rho(s,t,n)) \end{array}$$

The RHSs are morally 'simpler' than the LHSs. This can be made formal:

Theorem (Strong normalisation and confluence)

1. *The relation $\rhd$ is **terminating**: there is no infinite sequence $t_0 \rhd t_1 \rhd t_2 \rhd \cdots$.*
2. *Moreover $\rhd$ is **confluent**: there is a unique normal form u s.t. $t \rhd \cdots \rhd u$.*

Reduction and termination

The axioms we gave for the constants can be seen as a form of computation.

Orienting them left-right gives us the following notion of reduction:

$$\begin{array}{rcl} K_{\rho,\sigma}(s,t) & \triangleright & s \\ S_{\rho,\sigma,\tau}(s,t,u) & \triangleright & s(u)(t(u)) \end{array} \qquad \begin{array}{rcl} R_\rho(s,t,0) & \triangleright & g \\ R_\rho(s,t,\mathsf{s}(n)) & \triangleright & t(n,R_\rho(s,t,n)) \end{array}$$

The RHSs are morally 'simpler' than the LHSs. This can be made formal:

Theorem (Strong normalisation and confluence)

1. *The relation* $\triangleright$ *is* ***terminating****: there is no infinite sequence* $t_0 \triangleright t_1 \triangleright t_2 \triangleright \cdots$.
2. *Moreover* $\triangleright$ *is* ***confluent****: there is a unique normal form* u *s.t.* $t \triangleright \cdots \triangleright u$.

This result is not at all trivial! As we will see later, it implies the consistency of arithmetic. Usual proof methods are impredicative.

The Curry-Howard correspondence

The basic term calculus for system T, together with the reduction rules, is intimately related to logic.

The Curry-Howard correspondence

The basic term calculus for system T, together with the reduction rules, is intimately related to logic.

Theorem (Curry-Howard, informally)

1. *Each term t of type ρ describes a natural deduction proof of ρ.*
2. *$\triangleright$ induces an associated normalisation procedure on natural deduction proofs.*
3. *Terms in normal form describe the normal natural deduction proofs, i.e. those with no detours.*

NB: see [Troelstra and Schwichtenberg, 1996] for an excellent introduction to this area.

The Curry-Howard correspondence

The basic term calculus for system T, together with the reduction rules, is intimately related to logic.

Theorem (Curry-Howard, informally)

1. *Each term t of type ρ describes a natural deduction proof of ρ.*
2. *$\triangleright$ induces an associated normalisation procedure on natural deduction proofs.*
3. *Terms in normal form describe the normal natural deduction proofs, i.e. those with no detours.*

NB: see [Troelstra and Schwichtenberg, 1996] for an excellent introduction to this area.

Computation can also be related to cut-elimination, although the link is less straightforward. The result above is striking in its simplicity.

Outline

1. Functions and computability
2. Primitive recursion
3. Higher types: first steps
4. System T
5. Primitive recursion and beyond in System T
6. Some metalogical properties of system T
7. Approaches via well-founded orders
8. References

Avoiding higher types

While higher-order computation is incredibly powerful, it can be unintuitive.

It is worth taking a moment to explain that this is *not the only way* to express programs definable in arithmetic!

Avoiding higher types

While higher-order computation is incredibly powerful, it can be unintuitive.

It is worth taking a moment to explain that this is *not the only way* to express programs definable in arithmetic!

A promising alternative is by recursion on more complex data structures:

Definition (Well-founded orders)

A binary relation $\prec$ on $\mathbb{N}$ is a *well-order* if:

- If $x \prec y$ and $y \prec z$ then $x \prec z$.
- $x \prec y$ or $x = y$ or $y \prec x$.
- For every set $X \subseteq \mathbb{N}$, there is some $x \in X$ s.t. $\forall y \in X.(x = y \vee x \prec y)$.

(For convenience we will typically assume that 0 is always $\prec$-least in $\mathbb{N}$.)

Let $\prec$ be a well-order on $\mathbb{N}$.

Definition (Recursion on a well-order)

A function f is defined by *recursion on* $\prec$ from functions $g, h, \vec{k}$ if:

$$\begin{array}{lcll} f(0, \vec{x}) & = & g(\vec{x}) & \\ f(x, \vec{x}) & = & h(x, \vec{x}, f(k_1(x, \vec{x}), \vec{x}), \ldots f(k_n(x, \vec{x}), \vec{x})) & \text{if } x \neq 0 \end{array}$$

as long as, for each i, $\forall x. k_i(x, \vec{x}) \prec x$.

A glimpse of ordinals

Let $\prec$ be a well-order on $\mathbb{N}$.

Definition (Recursion on a well-order)

A function f is defined by *recursion on* $\prec$ from functions $g, h, \vec{k}$ if:

$$\begin{array}{lcll} f(0, \vec{x}) & = & g(\vec{x}) & \\ f(x, \vec{x}) & = & h(x, \vec{x}, f(k_1(x, \vec{x}), \vec{x}), \ldots f(k_n(x, \vec{x}), \vec{x})) & \text{if } x \neq 0 \end{array}$$

as long as, for each i, $\forall x. k_i(x, \vec{x}) \prec x$.

Notice the similarity to primitive recursion. It can be seen as a 'transfinite version' of what is known as *course-of-values* recursion.

A glimpse of ordinals

Let $\prec$ be a well-order on $\mathbb{N}$.

Definition (Recursion on a well-order)

A function f is defined by *recursion on* $\prec$ from functions $g, h, \vec{k}$ if:

$$\begin{array}{lcll} f(0, \vec{x}) & = & g(\vec{x}) & \\ f(x, \vec{x}) & = & h(x, \vec{x}, f(k_1(x, \vec{x}), \vec{x}), \ldots f(k_n(x, \vec{x}), \vec{x})) & \text{if } x \neq 0 \end{array}$$

as long as, for each i, $\forall x. k_i(x, \vec{x}) \prec x$.

Notice the similarity to primitive recursion. It can be seen as a 'transfinite version' of what is known as *course-of-values* recursion.

These orders can be embedded into a generalisation of of $\mathbb{N}$, known as the *ordinals*.

$$0\,1\,2\,\ldots\,\omega\,\omega+1\,\ldots\,2\omega\,2\omega+1\,\ldots\,\cdots\,\ldots\,\omega^2\,\ldots\,\cdots\,\cdots\,\omega^\omega\,\cdots\cdots\cdots\,\omega^{\omega^{\omega^{\cdots}}}$$

Computing the Ackermann-Péter function via recursion up to ω^ω

Define a well-order on pairs of numbers as follows:

$$\langle x, y\rangle \prec \langle x', y'\rangle \text{ if } x < y \text{ or } x = y \text{ and } x' < y'$$

We may construe the Ackermann-Péter function as a program on pairs, as follows:

$$\begin{aligned} A\langle 0, n\rangle &= n+1 \\ A\langle m+1, 0\rangle &= A\langle m, 1\rangle \\ A\langle m+1, n+1\rangle &= A\langle m, A\langle m+1, n\rangle\rangle \end{aligned}$$

Computing the Ackermann-Péter function via recursion up to ω^ω

Define a well-order on pairs of numbers as follows:

$$\langle x, y\rangle \prec \langle x', y'\rangle \text{ if } x < y \text{ or } x = y \text{ and } x' < y'$$

We may construe the Ackermann-Péter function as a program on pairs, as follows:

$$\begin{aligned} A\langle 0, n\rangle &= n+1 \\ A\langle m+1, 0\rangle &= A\langle m, 1\rangle \\ A\langle m+1, n+1\rangle &= A\langle m, A\langle m+1, n\rangle\rangle \end{aligned}$$

Notice that this is, indeed, an instance of recursion on $\prec$, since:

$$\begin{aligned} \langle m+1, 0\rangle &\succ \langle m, 1\rangle \\ \langle m+1, n+1\rangle &\succ \langle m+1, n\rangle \\ \langle m+1, n+1\rangle &\succ \langle m, A\langle m+1, n\rangle\rangle \end{aligned}$$

Computing the Ackermann-Péter function via recursion up to ω^ω

Define a well-order on pairs of numbers as follows:

$$\langle x, y\rangle \prec \langle x', y'\rangle \text{ if } x < y \text{ or } x = y \text{ and } x' < y'$$

We may construe the Ackermann-Péter function as a program on pairs, as follows:

$$\begin{aligned} A\langle 0, n\rangle &= n+1 \\ A\langle m+1, 0\rangle &= A\langle m, 1\rangle \\ A\langle m+1, n+1\rangle &= A\langle m, A\langle m+1, n\rangle\rangle \end{aligned}$$

Notice that this is, indeed, an instance of recursion on $\prec$, since:

$$\begin{aligned} \langle m+1, 0\rangle &\succ \langle m, 1\rangle \\ \langle m+1, n+1\rangle &\succ \langle m+1, n\rangle \\ \langle m+1, n+1\rangle &\succ \langle m, A\langle m+1, n\rangle\rangle \end{aligned}$$

NB: Notice the *impredicativity* of the final case.

Computing the Ackermann-Péter function via recursion up to ω^ω

Define a well-order on pairs of numbers as follows:

$$\langle x, y\rangle \prec \langle x', y'\rangle \text{ if } x < y \text{ or } x = y \text{ and } x' < y'$$

We may construe the Ackermann-Péter function as a program on pairs, as follows:

$$\begin{aligned} A\langle 0, n\rangle &= n+1 \\ A\langle m+1, 0\rangle &= A\langle m, 1\rangle \\ A\langle m+1, n+1\rangle &= A\langle m, A\langle m+1, n\rangle\rangle \end{aligned}$$

Notice that this is, indeed, an instance of recursion on $\prec$, since:

$$\begin{aligned} \langle m+1, 0\rangle &\succ \langle m, 1\rangle \\ \langle m+1, n+1\rangle &\succ \langle m+1, n\rangle \\ \langle m+1, n+1\rangle &\succ \langle m, A\langle m+1, n\rangle\rangle \end{aligned}$$

NB: Notice the *impredicativity* of the final case.

We may actually compute all well-definable functions of arithmetic by recursion on generalised versions of this well-order.

Outline

1. Functions and computability
2. Primitive recursion
3. Higher types: first steps
4. System T
5. Primitive recursion and beyond in System T
6. Some metalogical properties of system T
7. Approaches via well-founded orders
8. References

References I

Avigad, J. and Feferman, S. (1998).
Gödel's functional ("Dialectica") interpretation.
In Buss, S. R., editor, *Handbook of Proof Theory*, volume 137, pages 337–405. Elsevier.
http://www.andrew.cmu.edu/user/avigad/Papers/dialect.pdf.

Bimbó, K. (2018).
Combinatory logic.
In Zalta, E. N., editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, fall 2018 edition.

Boolos, G. S. and Jeffrey, R. C. (1989).
Computability and Logic: 3rd Ed.
Cambridge University Press, New York, NY, USA.

Gödel, K. (1958).
Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes.
Dialectica, 12:280–287.

Kohlenbach, U. (2008).
Applied Proof Theory - Proof Interpretations and their Use in Mathematics.
Springer Monographs in Mathematics. Springer.

References II

Troelstra, A. S. and Schwichtenberg, H. (1996).
Basic Proof Theory.
Cambridge University Press, New York, NY, USA.