CS 671, Automated Reasoning

Lesson 2: *What is a type? (I)* January 23, 2001

Type theory, as mentioned last time, is a theory that shall provide a foundation for computational mathematics. Because of that, we cannot express in terms of other theories, but have to keep it self-contained, that is we will describe it as a logic whose basic constructs directly express the intuition behind certain mathematical and computational concepts.

To keep things simple, let us begin with a concept that we are all familiar with, the *natural numbers*. How can we make this concept precise?

Typically we may think of the natural numbers as a *collection of elements*, say 0, 1, 2, 3, ... But there is more to the natural numbers than just that because we view these elements also as being related in some way.

- For instance, 1 is the *successor of* 0, 2 of 1, 3 of 2, etc.
- We know that we can *add two natural numbers* and get another one. We know that we can subtract, multiply, and divide and do many more things with natural numbers.
- We even know that we can *define* addition in terms of the more primitive successor operation, because we know that the sum n+0 of a number n and 0 is n, and that the sume of n and the successor of another number m is the same as the successor of n+m.

So we see that there are *several ways of denoting* natural numbers. One is writing down decimal representations 0, 1, 2, 3, ... Another is writing more complex expressions that involve *operations*, like 2+3, 4∗5, 36−(3∗(5+4)), ... A third is using *abstract placeholders*, like n, m, i, or *this-is-an-arbitrary-natural-number*.

The first method is the most direct one – we describe the *basic elements*, or *values* that make up the natural numbers. The second one consists of *compound expressions*, which we may *evaluate* to a primitive value, e.g. we may evaluate $36-(3*(5+4))$ to 9. The third one is symbolic – we use it to describe certain more general truths abstractly. So we might say:

> *The natural numbers are defined by their basic elements, by operations on these elements, and by a notion of evaluation of expressions consisting of elements and operations.*

This is a first step. Now the crucial thing is that we want to define this formally and that we want to do so using a methodology that may generalize beyond the natural numbers. As we do so, we want to make sure that our formal language remains reasonably small, that is whenever possible, we *only introduce elements and operations that cannot be expressed in terms of others*. All other operations and elements can be considered as *abbreviations* for expressions over these more primitive constructs.

For the natural numbers that means, we only take the number *zero* as primitive, since all other numbers can be expressed as successors. We only consider the *successor* operation as primitive and the principle of defining operations by *induction*. These are the only primitive constructs we will use.

Our methodology for formally defining the type of natural numbers will follow a principle first developed by the Swedish mathematician Per Martin-Löf. It consists of four basic steps:

1. Introducing the *syntax* of an unambiguous *formal language* that we will use to denote whatever we define.

   This is an absolute must if we want a computer to be able to "understand" what we're talking about.

2. Introducing a notion of *evaluation* on expressions, as a means to determine their value.

   This is necessary whenever we want to express computational concepts.

3. Defining the *semantics* of the type and its elements in terms of the above two concepts.

   This is necessary to assign meaning to pieces of formal text. Doing so in this particular way is one of the truly distinguishing features of Martin-Löf́s approach. It is the only way to define type theory independently of any other mathematical theory, and thus to consider it as a foundational theory (like set theory) instead of a derived one.

4. Introducing a *proof theory* that expresses the basic laws about the type and its elements in terms of syntactical *proof rules* that reflect the semantics.

   This is necessary if we want to do automated reasoning, since we want to express proofs as syntactical manipulations of formal text that we could perform with the aid of a computer.

It should be noted that this methodology is the result of a long succession of approaches to defining foundational logical theories. Martin-Löf was the first to properly integrate a computer scientist's view right into logic and to base the notion of semantics on *evaluation*.

Syntax

Our formal language for describing natural numbers needs 3 kinds of *expressions*

1. *Variables* `i, j, k, l, m, n, x, y, z,` ..., to denote arbitrary natural numbers
2. (At least) two *constants* 0, $\mathbb{N}$, which will receive a fixed meaning
3. *Compound expressions*, built from the above and the expressions `suc(`e`)` and `ind(`e`;` $base$`;` x`.`up`)`, where e, *base*, and *up* are expressions and x a variable that may occur in *up*.

The above definition of expressions is inductive and could to be written down a bit more formally, but we will postpone that to later. Note that e, *base*, x, and *up* are placeholders (or *meta-variables*) for arbitrary expressions and variables, but not part of the formal language itself. We need to make sure that we keep the *object language* of expressions separate separate from the *metalanguage* that we use for describing them

So the expressions we can form, are `0, suc(0), suc(suc(0)), ..., ind(0;0;x.x), ind(n;m;x.suc(x)), ...` but also `suc(`$\mathbb{N}$`), ind(n;m;x.`$\mathbb{N}$`)`, etc. Not all these expressions make sense, but that is an issue of semantics, not of syntax.

This language is somewhat minimalistic but that makes the later parts easier. Once we have understood this small theory properly, we can still declare certain other components (like decimal constants or arithmetic operations $+, -, *, /$) as primitive and describe their evaluation, semantics, and proof rules as part of the basic theory.

Evaluation

Some of the expressions that we just defined have an obvious meaning. 0 stands for zero, $\mathbb{N}$ for the type of natural numbers, `suc(0)` for one, `suc(suc(0))` for two, etc. But what about `ind(0; 0; x.x)`? What number is denoted by that?

To define this, we must introduce the concept of *evaluation*. Certain expressions, as we have seen, denote basic *values*, while others must be evaluated to determine what they denote. We sometimes call the former *canonical expressions* and the latter *noncanonical expressions*. To make things easier, we denote the evaluation of an expression e_1 into an expression e_2 by "$e_1 \downarrow e_2$".

Obviously most expressions of our little language evaluate to themselves, because they are already values:

$$\texttt{0} \downarrow \texttt{0},\quad \mathbb{N} \downarrow \mathbb{N},\quad \texttt{suc(0)} \downarrow \texttt{suc(0)},\quad \texttt{suc(suc(0))} \downarrow \texttt{suc(suc(0))}, \ldots$$

But the `ind` term evaluates to something different than itself. Intuitively, a term like $\texttt{ind(n;}\,base\texttt{;}\,x\texttt{.}up(x)\texttt{)}$ stands for some inductively defined value, that is for the result of applying a function f to n where

$$f(n) = \begin{cases} base & \text{if } n = 0 \\ up(f(m)) & \text{if } n = \texttt{suc}(m) \end{cases}$$

To express the evaluation of `ind` terms, we therefore define

1. $\texttt{ind(0;}\,base\texttt{;}\,x\texttt{.}up\texttt{)}$ evaluates to the value of *base*. Actually, whenever e evaluates to 0 then $\texttt{ind(}e\texttt{;}\,base\texttt{;}\,x\texttt{.}up\texttt{)}$ evaluates to the value of *base*.

   We use the following notation for that: $\dfrac{e \downarrow 0 \qquad\qquad base \downarrow val}{\texttt{ind(}e\texttt{;}\,base\texttt{;}\,x\texttt{.}up(x)\texttt{)} \downarrow val}$

2. To evaluate $\texttt{ind(suc(}e\texttt{);}\,base\texttt{;}\,x\texttt{.}up\texttt{)}$, we simply take the the term $\texttt{ind(}e\texttt{;}\,base\texttt{;}\,x\texttt{.}up\texttt{)}$, substitute every occurrence of x in up with that term, and then evaluate the modified expression. Actually, we can generalize that rule in the same way as before.

   $$\dfrac{e \downarrow \text{suc}(e') \qquad up[\texttt{ind(}e'\texttt{;}\,base\texttt{;}\,x\texttt{.}up\texttt{)}/x] \downarrow val}{\texttt{ind(}e\texttt{;}\,base\texttt{;}\,x\texttt{.}up\texttt{)} \downarrow val}$$

Note that we just introduced another "meta"-notation: $e[e'/x]$ denotes the expression that results from replacing every occurrence of the variable x in the expression e by e'. This *substitution* is a purely syntactical concept - we only replace text by text. In the literature you will find plenty of different notations for that like $e[x\backslash e'], e\{e'/x\}, e\{x\backslash e'\}, e_x^{e'}$ and many more. So you need to make sure that you know what the author means by these symbols.

$\boxed{\text{Q:}}$ *What is the result of evaluating* `ind(suc(suc(0));m;x.suc(x))` *?*

whenever we use concrete constants, as in the second example, we can evaluate and get `suc(ind(suc(0);m;x.suc(x)))`

There is still one unresolved question. What do we do with expressions of the form $\texttt{suc(ind(}e\texttt{;}\,base\texttt{;}\,x\texttt{.}up(x)\texttt{))}$? There are two ways to deal with them.

$$\texttt{0} \downarrow \texttt{0}, \quad \mathbb{N} \downarrow \mathbb{N}, \quad \texttt{suc(}e\texttt{)} \downarrow \texttt{suc(}e\texttt{)}$$

$$\frac{e \downarrow 0 \qquad\qquad \mathit{base} \downarrow \mathit{val}}{\texttt{ind(}e\texttt{;}\ \mathit{base}\texttt{;}\ x\texttt{.}\mathit{up}(x)\texttt{)} \downarrow \mathit{val}}$$

$$\frac{e \downarrow \mathrm{suc}(e') \quad \mathit{up}[\texttt{ind(}e'\texttt{;}\ \mathit{base}\texttt{;}\ x\texttt{.}\mathit{up}\texttt{)}/x] \downarrow \mathit{val}}{\texttt{ind(}e\texttt{;}\ \mathit{base}\texttt{;}\ x\texttt{.}\mathit{up}\texttt{)} \downarrow \mathit{val}}$$

Table 1: Evaluation rules for natural numbers

1. We could be *eager* and define that the value of that term is the successor of the value of `ind(`e`;` base`;` x`.`$\mathit{up}(x)$`)`. That is, our evaluation goes inside the terms and proceeds until every subterm has been evaluated. This is what programming languages name *call-by-value*.

2. We could be *lazy* and define that the expression evaluates to itself – the outer `suc` simply blocks any further evaluation until we actually need the value of the term in some other evaluation. Evaluation does not always lead to a primitive value.

   This is about what programming languages call this *call-by-name* and in some cases there are certain computational advantages of that approach.

Note that under eager evaluation the second rule for evaluating `ind` terms has the same effect as the following, which first evaluates e down to some value `suc(`v`)` (in two steps), then evaluates `ind(`v`;` base`;` x`.`up`)`, replaces every occurrence of x in up with that value, and finally evaluates the modified term.

$$\frac{e \downarrow \mathrm{suc}(e') \quad e' \downarrow v \quad \texttt{ind(}v\texttt{;}\ \mathit{base}\texttt{;}\ x\texttt{.}\mathit{up}\texttt{)} \downarrow v' \quad \mathit{up}[v'/x] \downarrow \mathit{val}}{\texttt{ind(}e\texttt{;}\ \mathit{base}\texttt{;}\ x\texttt{.}\mathit{up}(x)\texttt{)} \downarrow \mathit{val}}$$

Under lazy evaluation, the second rule for evaluating `ind` terms does much less. NUPRL's type theory essentially uses lazy evaluation,[1] for reasons that we will discuss later in the course. Table 1 summarizes all the evaluation rules for natural numbers that we will use.

$\boxed{\text{Q:}}$ *What happens to expressions like* `suc(`$\mathbb{N}$`)`, `ind(suc(0);m;x.`$\mathbb{N}$`)`, `ind(n;m;x.suc(x))`, *or* `ind(`$\mathbb{N}$`;m;x.x)`*?*

The first evaluates to itself under lazy evaluation, the second to $\mathbb{N}$, while evaluation is not defined for the other two, since we have variables or the constant $\mathbb{N}$ in a slot that expects expressions evaluating to number constants. An evaluation mechanism would usually abort at this point. So you may try to evaluate any expression of the formal language but you will not always get a value.

The reason not to define evaluation for all syntactically possible expressions of the formal language is that this way we make sure that we can later extend the theory and assign special meaning to certain terms. If evaluation were already defined for these terms – for instance as

[1]This is not entirely true. Arithmetical evaluation itself is eager: `8/4 + 5*3` $\downarrow$ `17`.
However `suc(8/4 + 5*3)` $\downarrow$ `suc (8/4 + 5*3)`. Use the ML function `compute` for tests.

resulting in some default value – we would have to *remove* rules from our theory before we can define new ones. Usually this means that some good properties of the theory may not be true anymore, since the evaluation of existing expressions has changed, and that we have to prove everything from scratch again. By leaving certain things undefined, the theory remains *open-ended*, which means we can add new concepts later and only have to investigate the properties of these additions and their consistency with the previous theory.

So the fact that certain expressions have no value, will not be expressed by the evaluation mechanism, but will be an implicit part of the semantics of the theory.

Remarks: