CS 671, Automated Reasoning

Lesson 4: *Primitive Recursive Functions in Type Theory* — January 30, 2001

Our formalization of a small theory of natural numbers has covered all the basic principles for building a formal type theory that can provide a foundation for computational mathematics and automated reasoning. We have discussed issues of syntax, evaluation, a semantics of judgments that is based on evaluation, and a proof theory that reflects this semantics.

Now that we have covered the basics, it is time to extend our small theory into something that is more expressive. Today, we want to add features that allow us to reason about *primitive recursive functions*. In the following lectures we will then add a theory of partial objects that enables us to reason about *all* computable functions and discuss Church's Thesis in this context.

There are three extensions that we need to make to be able to express primitive recursive functions. We need to make the `ind`-term more expressive so that it covers primitive recursion completely, we need to introduce reasoning about computation, and we need to formalize the concept of (total, computable) functions.

Making induction more expressive

The expression $\mathtt{ind}(n;\mathit{base};x.\mathit{up}(x))$ of our theory of natural numbers represents a very simple form of recursive definition. Roughly, it corresponds to applying a function f with

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

This form, however, is less expressiv than the usual primitive recursion, since it does not allow us to use the number n in the computations of the step case. We therefore cannot express simple functions like the predecessor function, whose recursive definition is

$$p(n) = \begin{cases} 0 & \text{if } n = 0 \\ m & \text{if } n = \mathtt{suc}(m) \end{cases}$$

The full form of primitive recursion is usually defined as follows.

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

It is easy to see that the predecessor function can now be represented.

$\boxed{\text{Q:}}$ *How do we have to change the syntax and evaluation of the* `ind` *term to accomodate this more general form?*

Obviously, we need to introduce an additional placeholder in our `ind` expression that allows the subterm *up* to refer to the number n. That means, we need to modify the `ind` term by adding a second variable that will be bound in *up*. Doing so affects our theory as follows:

Syntax: The inductive form will be extended to $\texttt{ind}(e\,;\,base\,;\,n\,,x\,.\,up)$

Evaluation: The evaluation rule for the base case remains almost unchanged. However, the evaluation rule for the step case has to take into account that the term up now refers to both the previous result and a "smaller" input.

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

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

Semantics: since $\texttt{ind}(e\,;\,base\,;\,n\,,x\,.\,up)$ is a noncanonical expression, there are no judgments that directly involve this term and the definition of the semantics is not affected by our modifications.

Proof Theory: The proof rule for the induction form now has to generate another declaration in its third subgoal to account for the assumption that n refers to a natural number.

4. $\bar{H} \vdash \texttt{ind}(e\,;\,base\,;\,n\,,x\,.\,up) \in \mathbb{N}$ **by** `ind-rule`
   $\bar{H} \vdash e \in \mathbb{N}$
   $\bar{H} \vdash base \in \mathbb{N}$
   $\bar{H}\,,\ n\!:\!\mathbb{N}\,,\ x\!:\!\mathbb{N} \vdash up \in \mathbb{N}$

The set of rules in our proof theory, however, is not yet complete. It does not allow us to prove that a term like `ind(suc(0); 0; n,x.suc(x))` actually denotes a natural number as the following example proof shows.

$\vdash$ `ind(suc(0); 0; n,x.suc(x))` $\in \mathbb{N}$ **by** `ind-rule`
1. $\vdash$ `suc(0)` $\in \mathbb{N}$ **by** `suc-rule`
   $\vdash$ `0` $\in \mathbb{N}$ **by** `0-rule` $\surd$
2. $\vdash$ `0` $\in \mathbb{N}$ **by** `0-rule` $\surd$
3. `n`:$\mathbb{N}$, `x`:$\mathbb{N}$ $\vdash$ `suc(x)` $\in \mathbb{N}$ **by** `suc-rule`
   `n`:$\mathbb{N}$, `x`:$\mathbb{N}$ $\vdash$ `x` $\in \mathbb{N}$ **by** `????`

We cannot complete the proof, since we lack a rule that relates declarations to the conclusion. Obviously, we can conclude that an expression x that is declared to be a natural number, actually is one. This is expressed by the following *structural* proof rule.

5. $\bar{H}_1\,,\ x\!:\!\mathbb{N}\,,\ \bar{H}_2 \vdash x \in \mathbb{N}$ **by** `hypothesis`

Reasoning about Equality and Computation

In order to be able to reason about computation and its results we have to extend our basic judgments by a notion of equality and introduce proof rules that represent the evaluation rules within our proof theory. So far there were two kinds of judgments that we could make

1. T is a type, *expressed in our proof theory as* T `type`
2. e is an element of T, *expressed in our proof theory as* $e \in T$

Extending our theory by a notion of equality means that we must be able to make judgments about two terms e and e' being equal. However, it is not sufficient to introduce a new judgment of the form

e and e' are equal

because the notion of equality does not only depend on the values of the two expressions. The numbers 2 and 5, for instance, are certainly not equal if we just look at them as natural numbers. Within the class $\mathbb{N}$ `mod 3`, however, 2 and 5 do denote the same value. Thus the *equality* of e and e' *depends on the type* to which e and e' belong. In other words, equality is a relation between three objects and the equality judgment needs to be formulated as:

e and e' are equal elements of T *which we express in our proof theory as* e=$e' \in T$

This judgment implicitly states that both e and e' must be elements of the type T and thus subsumes the second kind of judgment. As a consequence e is an element of T *can be considered as abbreviation for* e and e are equal elements of T.

Doing so simplifies the formulation of a theory of natural numbers with equality, as we do not have to formalize these two concepts separately. Instead we can extend the four judgments of our theory of natural numbers and use equality where we used membership:

1. $\mathbb{N}$ is a type
2. 0 and 0 are equal elements of $\mathbb{N}$
3. `suc(`e`)` and suc(e') are equal elements of $\mathbb{N}$ if e and e' are
4. if $e{\downarrow}v$, $e'{\downarrow}v'$ and v and v' are equal elements of $\mathbb{N}$ then so are e and e'

Similarly, the sequents of our proof theory will now deal with conclusions of the form e=$e' \in T$ instead of $e \in T$ and consider the latter simply as a shorthand notation for e=$e \in T$. The proof rules have to be modified as follows.

1. $\bar{H} \vdash \mathbb{N}$ `type` **by** $\mathbb{N}$`-rule`
2. $\bar{H} \vdash$ `0=0` $\in \mathbb{N}$ **by** `0-rule`
3. $\bar{H} \vdash$ `suc(`e`)=suc(`e'`)` $\in \mathbb{N}$ **by** `suc-rule`
   $\bar{H} \vdash e$=$e' \in \mathbb{N}$
4. $\bar{H} \vdash$ `ind(`e`;`$base$`;`n`,`x`.`up`)` = `ind(`e'`;`$base'$`;`n'`,`x'`.`up'`)` $\in \mathbb{N}$ **by** `ind-rule`
   $\bar{H} \vdash e$=$e' \in \mathbb{N}$
   $\bar{H} \vdash base$=$base' \in \mathbb{N}$
   $\bar{H}$, n:$\mathbb{N}$, x:$\mathbb{N} \vdash up$=$up'[n,x\,/\,n',x'] \in \mathbb{N}$
5. $\bar{H}_1$, x:$\mathbb{N}$, $\bar{H}_2 \vdash x$=$x \in \mathbb{N}$ **by** `hypothesis`

Note that in the fourth rule we have to substitute n and x for n' and x' in up' to express both up and up' are instantiated with the same value. Note also, that we do not need transitivity, commutativity and substitution rules for equality, as we cannot make equality assumptions at this point.

The above rules enable us to prove two terms to be equal if they are structurally equal (or α-equal), that is identical up to renaming of variables. They do not, however allow us to prove the equality of two expressions that are *semantically equal*, because they denote the same basic value. For this purpose, we have to add the following two rules about the evaluation of the induction form.

6. $\bar{H} \vdash \mathtt{ind(0;}\, base\mathtt{;}\, n\mathtt{,}x\mathtt{.}up\mathtt{)} = e' \in \mathbb{N}$ **by** `compute 1`
   $\bar{H} \vdash base = e' \in \mathbb{N}$

7. $\bar{H} \vdash \mathtt{ind(suc(}e\mathtt{);}\, base\mathtt{;}\, n\mathtt{,}x\mathtt{.}up\mathtt{)} = e' \in \mathbb{N}$ **by** `compute 1`
   $\bar{H} \vdash up[\mathtt{e'}, \mathtt{ind(}e'\mathtt{;}\, base\mathtt{;}\, x\mathtt{.}n\mathtt{,}up\mathtt{)} \,/\, n, x] = e' \in \mathbb{N}$

The parameter "1" in the name of the rule `compute 1` indicates that the computation is to be performed on the first equand of the equality. To evaluate the second equand, one has to invoke the rule as `compute 2`:

6. $\bar{H} \vdash e' = \mathtt{ind(0;}\, base\mathtt{;}\, n\mathtt{,}x\mathtt{.}up\mathtt{)} \in \mathbb{N}$ **by** `compute 2`
   $\bar{H} \vdash e' = base \in \mathbb{N}$

7. $\bar{H} \vdash e' = \mathtt{ind(suc(}e\mathtt{);}\, base\mathtt{;}\, n\mathtt{,}x\mathtt{.}up\mathtt{)} \in \mathbb{N}$ **by** `compute 2`
   $\bar{H} \vdash e' = up[\mathtt{e'}, \mathtt{ind(}e'\mathtt{;}\, base\mathtt{;}\, x\mathtt{.}n\mathtt{,}up\mathtt{)} \,/\, n, x] \in \mathbb{N}$

Let us demonstrate the application of these rules by a small example proof.

```
m:ℕ ⊢ ind(suc(suc(0)); m; n,x.suc(x)) = suc(suc(m)) ∈ ℕ     by compute 1
 m:ℕ ⊢ suc(ind(suc(0); m; n,x.suc(x))) = suc(suc(m)) ∈ ℕ     by suc-rule
  m:ℕ ⊢ ind(suc(0); m; n,x.suc(x)) = suc(m) ∈ ℕ               by compute 1
   m:ℕ ⊢ suc(ind(0; m; n,x.suc(x))) = suc(m) ∈ ℕ               by suc-rule
    m:ℕ ⊢ ind(0; m; n,x.suc(x)) = m ∈ ℕ                         by compute 1
     m:ℕ ⊢ m = m ∈ ℕ                                             by hypothesis √
```

It should be noted that the proof theory so far only allows us to reason about expressions that correspond to natural number constants. There is only limited support for reasoning about variables that are declared to be natural numbers, since we do not yet have a rule for decomposing assumptions and declarations. Proving a goal like

`n:ℕ, m:ℕ ⊢ n+m = m+n ∈ ℕ`

(where `n+m` is an abbreviation for `ind(n; m; i,x.suc(x))`) is not possible with the above 7 proof rules, since one has to apply induction over `n` and `m` to complete the proof. However, currently it is also not possible to generate the above proof goal if one starts a proof with a *primitive sequent*, i.e. a sequent without hypotheses, while all primitive sequents that correspond to a semantically true statement can also be proven in our theory. We will introduce rules for analyzing assumptions after we discuss logical propositions and quantifiers.

Function Spaces

Function types are the key to reasoning about computational functions and one of the main pillars of type theory. They link type theory to the λ-calculus and thus enable use to import well-understood principles from the theory of programming in a natural way.

Informally, a type $S \rightarrow T$ denotes the type of all (total) computable functions from S to T. Functions can be defined by λ-abstraction $\lambda x.e$, which defines a function that on input x evaluates the expression e. Functions can be used by applying them to concrete expressions. The basic computation principle is *β-reduction*, meaning that the application of an abstraction $(\lambda x.e)\ e'$ can be reduced to the term $e[e'/x]$. Our formal theory will be extended as follows:

Syntax: Three new expressions represent the theory of function spaces:

$S{\to}T$, $\lambda x\,.\,e$, and $f\ \ e$

where f, e, S, and T and are expressions and x a variable that may occur in e. To disambiguate notation, we will occasionally add parentheses to expressions in our presentation.

Evaluation: λ-abstractions are canonical terms that do not have to be evaluated (lazy evaluation), while applicationss are not considered as values.

$$\lambda x\,.\,e \downarrow \lambda x\,.\,e \qquad\qquad \frac{f \downarrow \lambda x.e' \quad e'[e/x] \downarrow val}{f\ \ e \downarrow val}$$

Semantics: Two new judgments are needed to describe equality of functions and typehood of function types

5. $S{\to}T$ is a type if S and T are
6. $\lambda x\,.\,e$ and $\lambda x'\,.\,e'$ are equal elements of $S{\to}T$ if $e[v/x]$ and $e'[v'/x']$ are equal elements of T whenever v and v' are equal elements of S

In addition to that we need to generalize the fourth judgment of natural numbers, which links equality to evaluation, so that it can be applied to elements of arbitrary types.

4'. if T is a type, $e{\downarrow}v$, $e'{\downarrow}v'$ and v and v' are equal elements of T then so are e and e'

Proof Theory: Four new rules need to be added, three for the decomposition of the newly introduced constructs and the fourth for reasoning about computation.

8. $\bar{H} \vdash S{\to}T$ `type` **by** $\to$`-rule`
   $\bar{H} \vdash S$ `type`
   $\bar{H} \vdash T$ `type`
9. $\bar{H} \vdash \lambda x\,.\,e$ = $\lambda x'\,.\,e' \in S{\to}T$ **by** λ`-rule`
   $\bar{H}$, $x{:}S \vdash e = e'[x/x'] \in T$
   $\bar{H} \vdash S$ `type`
10. $\bar{H} \vdash f\ \ e$ = $f'\ \ e' \in T$ **by** `app-rule` $S{\to}T$
   $\bar{H} \vdash f{=}f' \in S{\to}T$
   $\bar{H} \vdash e{=}e' \in S$
11. $\bar{H} \vdash (\lambda x\,.\,e)\ \ e'$ = $e^* \in T$ **by** `compute 1`
   $\bar{H} \vdash e'[e/x]$ = $e^* \in T$

Note that in the λ-rule we have to introduce S `type` as second subgoal to make sure that the expression right of the colon in the new declaration of the first subgoal is in fact a type. The application rule needs $S{\to}T$ as parameter, since the type of f (and f') cannot be determined from the immediate context.

Primitive Recursive Functions

The combination of function types and the theory of natural numbers provides the foundation for formal reasoning about *all* primitive recursive functions. In the following we give a few examples of such functions:

1. Addition: `add n m = n+m` $\quad$ `add` $\equiv$ λ`n.`λ`m. ind(n; m; i,x.suc(x))`
2. Predecessor: `p n = n`$\dot{-}$`1` $\quad$ `p` $\equiv$ λ`n. ind(n; 0; i,x.i)`
3. Subtraction: `sub n m = n`$\dot{-}$`m` $\quad$ `sub` $\equiv$ λ`m. ind(n; m; i,x.p x)`
4. Multiplication: `mul n m = n*m` $\quad$ `mul` $\equiv$ λ`n.`λ`m. ind(n; 0; i,x.add m x)`
5. Exponential: `exp n m = m`$^{\mathtt{n}}$ $\quad$ `exp` $\equiv$ λ`n.`λ`m. ind(n; 1; i,x.mul m x)`
6. Faculty: `fac n = n! = 1*2*...*n` $\quad$ `fac` $\equiv$ λ`n. ind(n; 1; i,x.mul i x)`
7. Less: `less n m = 0 if n<m, 1 if n`$\geq$`m`
   `less` $\equiv$ λ`n.`λ`m. ind(sub n m; 0; i,x.1)`
8. Maximum: `max n m = m if n<m, 1 if n`$\geq$`m`
   `max` $\equiv$ λ`n.`λ`m. ind(sub n m; m; i,x.n)`
9. f-product: π `f n = f(0)*f(1)*...*f(n`$\dot{-}$`1)`
   π $\equiv$ λ`f.`λ`n. ind(n; 1; i,x.mul (f i) x)`

Note that the f-product is a higher-order function: it takes as input a function and a number.

Here is a proof that addition is an element of $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ (we abbreviate e=$e \in T$ by $e \in T$).

$\vdash$ `add` $\in \mathbb{N} \to \mathbb{N} \to \mathbb{N}$	by λ-rule	
1. `n:`$\mathbb{N}$ $\vdash$ λ`m. ind(n; m; i,x.suc(x))` $\in \mathbb{N} \to \mathbb{N}$	by λ-rule	
1.1. `n:`$\mathbb{N}$`, m:`$\mathbb{N}$ $\vdash$ `ind(n; m; i,x.suc(x))` $\in \mathbb{N}$	by `ind-rule`	
1.1.1. `n:`$\mathbb{N}$`, m:`$\mathbb{N}$ $\vdash$ `n` $\in \mathbb{N}$	by `hypothesis`	$\surd$
1.1.2. `n:`$\mathbb{N}$`, m:`$\mathbb{N}$ $\vdash$ `m` $\in \mathbb{N}$	by `hypothesis`	$\surd$
1.1.3. `n:`$\mathbb{N}$`, m:`$\mathbb{N}$`, i:`$\mathbb{N}$`, x:`$\mathbb{N}$ $\vdash$ `suc(x)` $\in \mathbb{N}$	by `suc-rule`	
1.1.3.1. `n:`$\mathbb{N}$`, m:`$\mathbb{N}$`, i:`$\mathbb{N}$`, x:`$\mathbb{N}$ $\vdash$ `x` $\in \mathbb{N}$	by `hypothesis`	$\surd$
1.2. `n:`$\mathbb{N}$ $\vdash$ $\mathbb{N}$ `type`	by $\mathbb{N}$`-rule`	$\surd$
2. $\vdash$ $\mathbb{N}$ `type`	by $\mathbb{N}$`-rule`	$\surd$

$\boxed{\text{Q}}$: *prove the following properties*

$\vdash$ `fac (suc(suc(0))) = suc(suc(0))` $\in \mathbb{N}$

$\vdash$ $\pi \in (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$

Remarks: