5.4 Excursion: Logical Framework LF

The logical framework LF [Harper et al.(1993)Harper, Honsell, and Plotkin] is system for defining logics and formal systems based on the dependently-typed lambda-calculus. It is also sometimes referred to as $\lambda\Pi$-calculus. From a proof-theoretic point of view, it is equivalent to the fragment of first-order logic consisting of universal quantification, implication, and base predicates. In fact, even universal quantification and implication are collapsed when we interpret it from a type-theoretic perspective! Hence, while the simply-typed lambda-calculus has base types (proposition) and function space (impliciations), the $\lambda\Pi$-calculus has base type families (predicates) and dependent function space (universal quantification).

While very compact and small, it is an expressive framework that allows us to elegantly encode formal systems and logics. In particular, it is rich enough to encode and represent our natural deduction rules for first-order logic inside it! In fact there is a one-to-one correspondance between what proofs are described "on-paper" in our proof system and the objects described using the encoding of the proof system in LF. This is an important and often hard to achieve property.

5.4.1 Grammar and Typing

The logical framework LF allows us to define new types. Just as types classify terms, we can classify types using kinds. Let us start with the terms in LF. To characterize only terms in normal form, we split them into normal terms and neutral terms. Normal terms are either lambda-abstraction or a neutral term R. Neutral terms are either variable, constant, or an application of a neutral term to a normal term.

Kind	K	$::=$	$\mathsf{type} \mid \Pi x{:}A.K$
Type	A, B	$::=$	$\mathsf{a}\ M_1 \ldots M_n \mid \Pi x{:}A.B$
Normal Term	M, N	$::=$	$\lambda x.M \mid R$
Neutral Term	R	$::=$	$x \mid \mathsf{c} \mid R\ M$
Signature	Σ	$::=$	$\cdot \mid \Sigma, \mathsf{a} : K \mid \Sigma \mathsf{c} : A$

Types are declared to have a kind. The simple types have kind type. For example, we might declare a new type nat by stating in a signature that nat : type. Similarly, we can declare types that depend on arguments which are called type families. Such type families correspond logically to predicates. For example, we might want to declare that the type family (predicate) even takes in a natural number. This can be done by defining in the signature: even : Πn:nat.type. The type family (predicate) lt that relates and compares two natural numbers, can be defined as: lt : Πn:nat.Πn:nat.type.

We simply write $A \to K$ instead of $\Pi x{:}A.K$, if x does not occur in the free in K. Hence, the previous three definitions can simply be stated as:

$$\begin{array}{lcl} \mathsf{nat} & : & \mathsf{type} \\ \mathsf{even} & : & \mathsf{nat} \to \mathsf{type} \\ \mathsf{lt} & : & \mathsf{nat} \to \mathsf{nat} \to \mathsf{type} \end{array}$$

Types correspond logically to propositions. They are formed using atomic types $a\ M_1 \ldots M_n$ where a stands for a type constant (or type family or logically a predicate) and $M_1 \ldots M_n$ are the arguments. How many arguments a type constant accepts is determined by its kind. For example, the type constant nat is declared simply of kind type and takes in no arguments; the type constant even takes in a natural number as an argument; the type constant lt takes in two natural numbers.

We can declare in the signature term constants together with their type. Here are a few examples:

$$\begin{array}{lcl} \mathsf{z} & : & \mathsf{nat}. \\ \mathsf{s} & : & \mathsf{nat} \to \mathsf{nat}. \end{array}$$

We again write simply $A \to B$ instead of $\Pi x{:}A.B$ if x does not occur in B. Hence, the dependent function space, called Π-type, degenerates to the simply typed function space $A \to B$.

$$\begin{array}{lcl} \mathsf{ev\_z} & : & \mathsf{even\ z}. \\ \mathsf{ev\_s} & : & \Pi \mathsf{N{:}nat.even\ N} \to \mathsf{even\ (s(s\ N))}. \end{array}$$

Let us show the typing rules. We write P or Q for atomic types.

$\boxed{\Gamma \vdash M \Leftarrow A}$ Normal Term M checks against type A.

$$\frac{\Gamma, x{:}A \vdash M \Leftarrow B}{\Gamma \vdash \lambda x.M \Leftarrow \Pi x{:}A.B}\ \mathsf{abs} \qquad \frac{\Gamma \vdash R \Rightarrow a\ M_1 \ldots M_n}{\Gamma \vdash R \Leftarrow A\ M_1 \ldots M_n}\ \mathsf{coe}$$

$\boxed{\Gamma \vdash R \Rightarrow A}$ Neutral Term R synthesizes type A.

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A}\ \mathsf{var} \qquad \frac{c : A \in \Sigma}{\Gamma \vdash c \Rightarrow A}\ \mathsf{const} \qquad \frac{\Gamma \vdash R \Rightarrow \Pi x{:}A.B \qquad \Gamma \vdash M \Leftarrow A}{\Gamma \vdash R\ M \Rightarrow [M/x]B}\ \mathsf{app}$$

The typing rules should look very familiar, as they correspond to the annotated normal proofs for the fragment of first order logic that only considers universal quantifiers and implication, where we collapse the rules for implications and universal quantification.

Remark In the rule for application, we replace x by M in the type B. In general, this could lead to a type that is ill-formed in our grammar. Consider for example, $B = a\ (x\ z)$. If we replace x by $\lambda y.y$, then we obtain $[\lambda y.y/x](a\ (x\ z)) = a\ ((\lambda y.y)\ z)$. The result is however not a normal term anymore and hence, according to our grammar, ill-formed.

We hence emplay *hereditary substitution* operation which replaces x by M and eliminates all possible redeces as we go along. This operation goes back to [Watkins et al.(2002)Watki
We omit the precise definition of *hereditary substitution* here, but refer the reader to for example [Cave and Pientka(2012)] or similar work.

Finally, we now consider what we can do within this framework. Besides specifying natural numbers and properties such as even, we can use it to check derivations. For example, using the typing rules above, we can justify the following:

$$\mathsf{ev\_s}\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{z}))\ (\mathsf{ev\_s}\ \mathsf{z}\ \mathsf{ev\_z}) \quad : \quad \mathsf{even}\ (\mathsf{s}\ (\mathsf{s}\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{z}))))$$

$\mathcal{D}_1 =$

$$\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{s} \Rightarrow \mathsf{nat} \to \mathsf{nat}}\ \mathsf{const} \quad \dfrac{\dfrac{\dfrac{}{\vdash \mathsf{z} \Rightarrow \mathsf{nat}}\ \mathsf{const}}{\vdash \mathsf{z} \Leftarrow \mathsf{nat}}\ \mathsf{coe}}{\vdash \mathsf{s\ z} \Rightarrow \mathsf{nat}}\ \mathsf{app}}{\vdash \mathsf{s\ z} \Leftarrow \mathsf{nat}}\ \mathsf{coe} \quad \dfrac{\dfrac{\dfrac{}{\vdash \mathsf{s} \Rightarrow \mathsf{nat} \to \mathsf{nat}}\ \mathsf{const} \quad \dfrac{\dfrac{\dfrac{}{\vdash \mathsf{s} \Rightarrow \mathsf{nat} \to \mathsf{nat}}\ \mathsf{const}}{\vdash \mathsf{s\ (s\ z)} \Rightarrow \mathsf{nat}}\ \mathsf{coe}}{\vdash \mathsf{s\ (s\ z)} \Leftarrow \mathsf{nat}}\ \mathsf{app}}{\vdash \mathsf{s\ (s\ (s\ z))} \Rightarrow \mathsf{nat}}\ \mathsf{coe}}{\vdash \mathsf{s\ (s\ (s\ z))} \Leftarrow \mathsf{nat}}\ \mathsf{app}$$

$\mathcal{D}_2 =$

$$\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{ev\_s} \Rightarrow \Pi \mathsf{n{:}nat.even\ n} \to \mathsf{even\ (s\ (s\ n))}}\ \mathsf{const} \quad \dfrac{\dfrac{\dfrac{}{\vdash \mathsf{z} \Rightarrow \mathsf{nat}}\ \mathsf{const}}{\vdash \mathsf{z} \Leftarrow \mathsf{nat}}\ \mathsf{coe}}{\vdash \mathsf{ev\_s\ z} \Rightarrow \mathsf{even\ z} \to \mathsf{even\ (s\ (s\ z))}}\ \mathsf{app} \quad \dfrac{\dfrac{\dfrac{}{\vdash \mathsf{ev\_z} \Rightarrow \mathsf{even\ z}}\ \mathsf{const}}{\vdash \mathsf{ev\_z} \Leftarrow \mathsf{even\ z}}\ \mathsf{coe}}{\vdash \mathsf{ev\_s\ z\ ev\_z} \Rightarrow \mathsf{even\ (s\ (s\ z))}}\ \mathsf{app}}{\vdash \mathsf{ev\_s\ z\ ev\_z} \Leftarrow \mathsf{even\ (s\ (s\ z))}}\ \mathsf{coe}$$

Using this derivation in the derivation below, we can justify :

$$\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{ev\_s} \Rightarrow \Pi \mathsf{N{:}nat.even\ N} \to \mathsf{even\ (s(s\ N))}}\ \mathsf{const} \quad \overset{\mathcal{D}_1}{\vdash \mathsf{s\ (s\ z))} \Leftarrow \mathsf{nat}}}{\vdash \mathsf{ev\_s\ (s\ (s\ z))} \Rightarrow \mathsf{even\ (s\ (s\ z))} \to \mathsf{even\ (s(s\ (s\ (s\ z))))}}\ \mathsf{app} \quad \overset{\mathcal{D}_2}{\vdash \mathsf{ev\_s\ z\ ev\_z} \Leftarrow \mathsf{even\ (s\ (s\ z))}}}{\vdash \mathsf{ev\_s\ (s\ (s\ z))\ (ev\_s\ z\ ev\_z)} \Rightarrow \mathsf{even\ (s(s\ (s\ (s\ z))))}}\ \mathsf{app}$$

As this example shows in detail, we can use the rules of the $\lambda\Pi$-calculus to encode our own theories by defining new types and type families as well as constants inhabiting them.

5.4.2 Beluga: a proof and programming environment based on the logical framework LF

Beluga [Pientka and Dunfield(2010), Pientka and Cave(2015)] provides an implementation of the logical framework LF. The previous example of natural numbers and definition of even, can be encoded in Beluga's concrete syntax as follows:

```
LF nat: type =
| z : nat
| s : nat → nat;

LF even: nat → type =
| ev_z : even z
| ev_s : even N → even (s (s N));
```

The keyword **LF** states we are introducing an LF type (family) together with constants of that type. For convenience, we omitted the universal quantifier (written using Π in the previous section), but by convention all free variables in a given type are universally quantified at the outside implicitly. Beluga's reconstruction engine will infer the type of `N` and abstract over `N` at the outside. We hence elaborate the type `even N → even (s (s N))` to Π`N:nat. even N → even (s (s N))`.

Beluga further has a type checker that verifies that a given term has a given type following the typing rules from the previous section. The type checker is clever enough that we can omit passing instantiations for `N` when we rely on the Π-elimination rule in our derivation. Intead of $\vdash$ ev_s z ev_z $\Leftarrow$ even (s (s z)) we simply write the following concrete expression in Beluga:

```
rec d1 : [ ⊢ even (s (s z))] = [ ⊢ ev_s ev_z];
```

We use **rec** as a keyword of introducing constants that stand for proofs. Note that we omit writing `z` and do not pass it as an argument to `ev_s` explicitly; however, type reconstruction in Beluga will infer it. This makes derivations more readable. We refer to LF objects inside `[ ]`. The turnstyle $\vdash$ is used to separate assumptions from the main statement we are trying to establish. In the given example, all derivations were closed and did not rely on hypothesis.

Let us look at a few more examples.

Encoding Addition We can encode addition as a type family relating three natural numbers as follows:

```
LF add: nat → nat → nat → type =
| a_z: add z N N
| a_s: add N M K → add (s N) M (s K);
```

We again omit writing Π-quantification over `N`, `M`, and `K` explicitly, and let Beluga's reconstruction engine infer the type of the free variables and abstract over them implicitly. Then we can state the proof of the fact that adding `s (s z)` to a variable `N` results in `s (s N)`.

```
rec a1 : [ ⊢ add (s (s z)) N (s (s N))] = [⊢ a_s (a_s a_z)];
```

Beluga enforces the use of capital letters for universally quantified variables. For example, the previous statement is for all `N` that are natural numbers.

Encoding Vectors We can encode boolean vectors as lists that are indexed by their length as follows using a type family `vec` that takes in `nat` as an argument. We then construct vectors either with the empty vector described by the constant `e` or by the constant `snoc` that takes in a vector of length `N` and a `bool` and constructs a vector of length `s N`.

```
LF bool : type =
| tt: bool
| ff: bool;

LF vec: nat → type =
| e: vec z
| snoc : vec N → bool → vec (s N);
```

We can then for example possible vectors of length `(s (s z))`.

```
rec v0 : [⊢ vec (s (s z))] = [⊢ snoc (snoc e tt) ff];
rec v1 : [⊢ vec (s (s z))] = [⊢ snoc (snoc e ff) ff];
rec v2 : [⊢ vec (s (s z))] = [⊢ snoc (snoc e ff) tt];
rec v3 : [⊢ vec (s (s z))] = [⊢ snoc (snoc e tt) tt];
```

Encoding Natural Deduction We now encode the natural deduction rules for propositional logic in LF. This will allow us to encode and check natural deduction proofs using the simple typing rules of LF. We first define a type `o` describing logical propositions we want to represent.

```
LF o : type =
| imp : o → o → o
| &   : o → o → o
| top : o;
```

Let's revisit our introduction and elimination rules for these propositions.

$\boxed{A\ \mathsf{true}}$ A is true

$$\frac{}{\top\ \mathsf{true}}\ \top\mathrm{I} \qquad \frac{A \supset B\ \mathsf{true} \qquad A\ \mathsf{true}}{B\ \mathsf{true}}\ \supset\mathrm{E} \qquad \frac{\begin{array}{c}\overline{A\ \mathsf{true}}\ u \\ \vdots \\ B\ \mathsf{true}\end{array}}{A \supset B\ \mathsf{true}}\ \supset\mathrm{I}^u$$

$$\frac{A\ \mathsf{true} \qquad B\ \mathsf{true}}{A \wedge B\ \mathsf{true}}\ \wedge\mathrm{I} \qquad \frac{A \wedge B\ \mathsf{true}}{A\ \mathsf{true}}\ \wedge\mathrm{E}_l \qquad \frac{A \wedge B\ \mathsf{true}}{B\ \mathsf{true}}\ \wedge\mathrm{E}_r$$

We represent the judgment A true using the type family `nd`. Each inference rule turns into a constant of the type family `nd`.

```
LF nd: o → type.
| andI : nd A → nd B → nd (& A B)
| andEl : nd (& A B) → nd A
| andEr : nd (& A B) → nd B
| impI : (nd A → nd B) → nd (imp A B)
| impE : nd (imp A B) → nd A → nd B
| topI : nd top;
```

Some of these constant definitions are straightforward. For example, the constant `andI` directly encodes the inference rule $\wedge$I. Similarly, the constant `andEl` and `andEr` correspond directly to the inference rules $\wedge\mathrm{E}_l$ and $\wedge\mathrm{E}_r$. The constant `topI` corresponds directly to the rule λ`top I`.

The most interesting is the encoding of the hypothetical sub-derivation in the rule $\supset$ I. How shall we encode the hypothetical derivation "given the hypothesis A true we must derive a proof of B true"? - Here is the trick: LF has function spaces. We map the hypothetical derivation "given the hypothesis A true we must derive a proof of B true" to the function `nd A → nd B`. This form of encoding is called higher-order abstract syntax as our abstract syntax trees may contain functions which we use to model the scope of hypothesis in hypothetical derivations. Functions in Beluga are written as `λu. M` where `M` is the body of the function and we are abstracting over the variable `u`.

Let's look at a few examples.

```
rec p0 : [ ⊢ nd (imp (& A B) A)]  =
  [ ⊢ impI (λu. andEl u)] ;
rec p1 : [ ⊢ nd (imp (& A B) (& B A))] =
  [ ⊢ impI λu. andI (andEr u) (andEl u)];
rec p2 : [ ⊢ nd (imp (& (imp A B) (imp B C)) (imp A C))] =
  [ ⊢ impI λu. (impI λv. impE (andEr u) (impE (andEl u)  v )) ];
```

The astute reader will observe that unlike Tutch where we write in each line the formula that can be derived from the previous lines, we here write only the justifications (i.e. `impI` or `andEl`), but not the formula itself. Beluga's type checker will make sure that the given justifications indeed constitute a valid proof. The scope of an assumption is introduced using an LF-function written as `λu.M`.

It is instructive to give different names to constants and type family and given them more suggestive names. Let's do the following replacement

Original name	Replacement
`nd`	`tm`
`andI`	`pair`
`andEl`	`fst`
`andEr`	`snd`
`impI`	`lam`
`impE`	`app`
`topI`	`unit`

This results in the equivalent definitions that might be more familiar to read:

```
LF tm : o → type =
| pair : tm A → tm B → tm (& A B)
| fst : tm (& A B) → tm A
| snd : tm (& A B) → tm B
| lam : (tm A → tm B) → tm (imp A B)
| app : tm (imp A B) → tm A → tm B
| unit : tm top;

rec t0 : [ ⊢ tm (imp (& A B) A)] =
  [ ⊢ lam (λu. fst u)] ;
rec t1 : [ ⊢ tm (imp (& A B) (& B A))] =
  [ ⊢ lam λu. pair (snd u) (fst u)];
rec t2 : [ ⊢ tm (imp (& (imp A B) (imp B C)) (imp A C))] =
  [ ⊢ lam λu. (lam λv. app (snd u) (app (fst u)  v )) ];
```

Here we view the definition of natural deduction proofs as a definition of well-typed terms.

As this discussion illustrates, the logical framework LF acts like a *universal* proof checker. We encode our logic, for example our rules about even numbers, addition, or our natural deduction rules, and then can encode derivations as LF objects and use type checking to verify that a given derivation is valid.

Chapter 6

Induction

So far, we have seen first-order logic together with its corresponding proof-terms. First-order logic corresponds to the dependently typed lambda-calculus. However, if we are to write meaningful programs we need two more ingredients: 1) we need to reason about specific domains such as natural numbers, lists, etc 2) we need to be able to write recursive programs about elements in a given domain. Proof-theoretically, we would like to add the power of induction which as it turns out corresponds to being able to writ total well-founded recursive programs.

6.1 Domain: natural numbers

First-order logic is independent of a given domain and the reasoning principles we defined hold for any domain. Nevertheless, it is useful to consider specific domains. There are several approaches to incorporating domain types or index types into our language: One is to add a general definition mechanism for *recursive types* or *inductive types*. We do not consider this option here, but we return to this idea later. Another one is to use the constructs we already have to define data. This was Church's original approach; he encoded numerals, booleans as well as operations such as addition, multiplication, if-statements, etc. as lambda-terms using a *Church encoding*. We will not discuss this idea in these notes. A third approach is to specify each type directly by giving rules defining how to construct elements of a given type (introduction rule) and how to reason with elements of a given type (elimination rule). This is the approach we will be pursuing here.

We begin by defining concretely the judgement

$$t : \tau \quad \text{Term } t \text{ has type } \tau$$

which we have left more or less abstract for now for concrete instances of τ.

6.1.1 Defining for natural numbers

We define elements belonging to natural numbers via two constructors z and suc. They allow us to introduce natural numbers. We can view these two rules as introduction rules for natural numbers.

$$\frac{}{\mathsf{z} : \mathsf{nat}}\ \mathsf{natI}_{\mathsf{z}} \qquad \frac{\mathsf{t} : \mathsf{nat}}{\mathsf{suc}\,\mathsf{t} : \mathsf{nat}}\ \mathsf{natI}_{\mathsf{suc}}$$

6.1.2 Reasoning about natural numbers

To prove inductively a property $A(\mathsf{t})\ \mathsf{true}$, we establish three things:

1. t is a natural number and hence we know how to split it into different cases.
2. *Base case*: $A(\mathsf{z})\ \mathsf{true}$
   Establish the given property for the number z
3. *Step case*: For any $\mathfrak{n} : \mathsf{nat}$, assume $A(\mathfrak{n})\ \mathsf{true}$ (I.H) and prove $A(\mathsf{suc}\,\mathfrak{n})\ \mathsf{true}$.
   We assume the property holds for smaller numbers, i.e. for $\mathfrak{n}$, and we establish the property for $\mathsf{suc}\,\mathfrak{n}$.

More formally, the inference rule capturing this idea is given below:

$$\frac{\mathsf{t} : \mathsf{nat} \qquad A(\mathsf{z})\ \mathsf{true} \qquad \begin{array}{c}\dfrac{}{\mathfrak{n} : \mathsf{nat}} \quad \dfrac{}{A(\mathfrak{n})\ \mathsf{true}}\ \mathfrak{i}.\mathfrak{h} \\ \vdots \\ A(\mathsf{suc}\,\mathfrak{n})\ \mathsf{true}\end{array}}{A(\mathsf{t})\ \mathsf{true}}\ \mathsf{natE}^{\mathfrak{n},\mathfrak{ih}}$$

Restating the rule using explicit contexts to localize all assumptions:

$$\frac{\Gamma \vdash \mathsf{t} : \mathsf{nat} \qquad \Gamma \vdash A(\mathsf{z})\ \mathsf{true} \qquad \Gamma,\ \mathfrak{n}{:}\mathsf{nat},\ \mathfrak{ih}{:}A(\mathfrak{n})\ \mathsf{true} \vdash A(\mathsf{suc}\,\mathfrak{n})\ \mathsf{true}}{\Gamma \vdash A(\mathsf{t})\ \mathsf{true}}\ \mathsf{natE}^{\mathfrak{n},\mathfrak{ih}}$$

Let us prove a simple property inductively, to see how we use the rule.

Note the rule $\mathsf{natE}^{\mathfrak{n},\mathfrak{ih}}$ has implicitly a generalization built-in. If we want to establish a property $A(42)\ \mathsf{true}$, we may choose to prove it more generally for any number

proving that $A(z)$ true (i.e. the property A holds for z) and proving $A(\mathsf{suc}\, n)$ true (i.e. the property A holds for $\mathsf{suc}\, n$) assuming the property A holds for n. Since we now have proven that A holds for all natural numbers, it also must hold for 42.

Let's look at an example; we first encode definitions which will form our signature $\mathcal{S}$.

$$\begin{array}{ll} \mathsf{le\_z} & : \forall x{:}\mathsf{nat}.\ z \leq x \\ \mathsf{le\_suc} & : \forall x{:}\mathsf{nat}.\forall y{:}\mathsf{nat}.\ x \leq y \ \supset\ \mathsf{suc}\, x \leq \mathsf{suc}\, y \end{array}$$

Then we would like to prove: $\mathcal{S} \vdash \forall x{:}\mathsf{nat}.\ x \leq x$ true. For better readability, we omit true in the derivations below.

$$\dfrac{\dfrac{\overline{\mathcal{S},\ a{:}\mathsf{nat} \vdash a : \mathsf{nat}} \qquad \mathcal{D}_z \qquad \mathcal{D}_{\mathsf{suc}}}{\mathcal{S},\ a{:}\mathsf{nat} \vdash a \leq a}\ \mathsf{natE}^{n,ih}}{\mathcal{S} \vdash \forall x{:}\mathsf{nat}.\ x \leq x}\ \forall I^{a}$$

We consider the base case $\mathcal{D}_z$ and the step case $\mathcal{D}_{\mathsf{suc}}$ separately. We also write the derivations using implicit context representations to keep them compact.

$$\mathcal{D}_z = \dfrac{\dfrac{}{\forall x{:}\mathsf{nat}.z \leq x}\ \mathsf{le\_z} \qquad \dfrac{}{z : \mathsf{nat}}\ \mathsf{nat}_z}{z \leq z}\ \forall E$$

$$\mathcal{D}_{\mathsf{suc}} = \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall x{:}\mathsf{nat}.\forall y{:}\mathsf{nat}.\ x \leq y \ \supset\ \mathsf{suc}\, x \leq \mathsf{suc}\, y}\ \mathsf{le\_suc} \qquad \dfrac{}{n : \mathsf{nat}}}{\forall y{:}\mathsf{nat}.\ n \leq y \ \supset\ \mathsf{suc}\, n \leq \mathsf{suc}\, y}\ \forall E \qquad \dfrac{}{n : \mathsf{nat}}}{n \leq n \supset \mathsf{suc}\, n \leq \mathsf{suc}\, n}\ \forall E \qquad \dfrac{}{n \leq n}\ ih}{\mathsf{suc}\, n \leq \mathsf{suc}\, n}\ \supset E$$

6.1.3 Proof terms

We assign a recursive program to the induction rule.

$$\dfrac{\Gamma \vdash t : \mathsf{nat} \qquad \Gamma \vdash M_z : A(z) \qquad \Gamma,\ n{:}\mathsf{nat},\ f\ n{:}A(n)\ \mathsf{true} \vdash M_{\mathsf{suc}} : A(\mathsf{suc}\, n)}{\Gamma \vdash \mathsf{rec}^{\forall x:\mathsf{nat}.A(x)}\ t\ \mathsf{with}\ f\ z \to M_z \mid f\ (\mathsf{suc}\, n) \to M_{\mathsf{suc}} : A(t)}\ \mathsf{natE}^{n,\, f\, n}$$

The proof term uses the variable f to denote the function we are defining; in some sense, this definition is similar to programs allowing defining functions by equations using simultaneous pattern as in Haskell.

From the proof above for $\mathcal{S} \vdash \forall x{:}\mathsf{nat}.\ x \leq x$ we can synthesize the following program:

$$\begin{array}{l}\lambda a{:}\mathsf{nat}.\ \mathsf{rec}^{\forall x:\mathsf{nat}.\ x\leq x}\ a\ \mathsf{with}\\ \quad \mid f\,z \quad\ \Rightarrow\ \mathsf{le\_z}\ z\\ \quad \mid f\,(\mathsf{suc}\,n) \ \Rightarrow\ \mathsf{le\_suc}\ n\ n\ (f\,n)\end{array}$$

Proving Predecessor We will next prove next $\forall x{:}\mathsf{nat}.\neg(x = z) \supset \exists y : \mathsf{nat}.\mathsf{suc}\,y = x$. For simplicity, we define equality using reflexivity: $\mathsf{ref} : \forall x{:}\mathsf{nat}.x = x$.

$$\dfrac{\begin{array}{c}\mathcal{D}\\ \mathcal{S}, a : \mathsf{nat} \vdash \neg(a = z) \supset \exists y : \mathsf{nat}.\mathsf{suc}\,y = a\end{array}}{\mathcal{S} \vdash \forall x{:}\mathsf{nat}.\neg(x = z) \supset \exists y : \mathsf{nat}.\mathsf{suc}\,y = x}\ \forall I^{a}$$

To prove $\mathcal{D}$ we will use the rule $\mathsf{natE}^{n,f\,n}$; in particular, we need to prove the following base and step case:
Base case:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\neg(z = z)}\,u \qquad \dfrac{}{z = z}\,\mathsf{ref}}{\bot}\supset E}{\exists y{:}\mathsf{nat}.\ \mathsf{suc}\,y = z}\bot E}{\neg(z = z) \supset \exists y{:}\mathsf{nat}.\ \mathsf{suc}\,y = z}\supset I^{u}$$

Step case:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{suc}\,(n) = \mathsf{suc}\,(n)}\,\mathsf{ref} \qquad \dfrac{}{n : \mathsf{nat}}}{\exists y{:}\mathsf{nat}.\ \mathsf{suc}\,(y) = \mathsf{suc}\,(n)}\exists I}{\neg(\mathsf{suc}\,(n) = z) \supset \exists y{:}\mathsf{nat}.\ \mathsf{suc}\,(y) = \mathsf{suc}\,(n)}\supset I^{u}$$

Therefore, we have established $\forall x{:}\mathsf{nat}.\neg(x = z) \supset \exists y : \mathsf{nat}.\mathsf{suc}\,y = x$. Note that we only used the induction rule to split the proof into two different cases, but we did not actually use the induction hypothesis in the proof.

It is particularly interesting to extract the corresponding proof term; we omit the type annotation on the rec for better readability.

$$
\begin{array}{l}
\lambda a{:}\mathsf{nat}.\ \mathsf{rec}\ a\ \mathsf{with} \\
\qquad \mid \mathsf{f}\,\mathsf{z} \quad\ \Rightarrow\ \lambda u{:}\neg(x = \mathsf{z}).\mathsf{abort}\ (u\ (\mathsf{refl}\ \mathsf{z})) \\
\qquad \mid \mathsf{f}\,(\mathsf{suc}\,n) \ \Rightarrow\ \lambda u{:}\neg(x = \mathsf{z}).\langle n,\ \mathsf{refl}\ \mathsf{z}\rangle
\end{array}
$$

Can you guess what program it implements? - If we erase all subterms pertaining propositions, it becomes even more obvious:

$$
\begin{array}{l}
\lambda a{:}\mathsf{nat}.\ \mathsf{rec}\ a\ \mathsf{with} \\
\qquad \mid \mathsf{f}\,\mathsf{z} \quad\ \Rightarrow\ \_\_ \\
\qquad \mid \mathsf{f}\,(\mathsf{suc}\,n) \ \Rightarrow\ n
\end{array}
$$

We have obtained a verified predecessor function!

How to extend our notion of computation? We will have two reduction rules which allow us to reduce a recursive program:

$$
\begin{array}{lll}
\mathsf{rec}^A\ \mathsf{z} & \mathsf{with}\ \mathsf{f}\ \mathsf{z} \to M_\mathsf{z} \mid \mathsf{f}\ (\mathsf{suc}\,n) \to M_\mathsf{suc} & \Longrightarrow \quad M_\mathsf{z} \\
\mathsf{rec}^A\ (\mathsf{suc}\,t) & \mathsf{with}\ \mathsf{f}\ \mathsf{z} \to M_\mathsf{z} \mid \mathsf{f}\ (\mathsf{suc}\,n) \to M_\mathsf{suc} & \Longrightarrow \quad [t/n][r/\mathsf{f}\,n]M_\mathsf{suc} \\
 & \qquad \text{where } r = \mathsf{rec}^A\ t\ \mathsf{with}\ \mathsf{f}\ \mathsf{z} \to M_\mathsf{z} \mid \mathsf{f}\ (\mathsf{suc}\,n) \to M_\mathsf{suc} &
\end{array}
$$

Note that we unroll the recursion by replacing the reference to the recursive call $\mathsf{f}\,n$ with the actual recursive program $\mathsf{rec}^A\ t\ \mathsf{with}\ \mathsf{f}\ \mathsf{z} \to M_\mathsf{z} \mid \mathsf{f}\ (\mathsf{suc}\,n) \to M_\mathsf{suc}$.

We might ask, how to extend our congruence rules. In our setting, were reductions can happen at any given sub-term, we will have two additional congruence rules. Note that we do not have a rule which evaluates t, the term we are recursing over; at the moment, the only possible terms we can have are those formed by z and suc or variables. We have no computational power on for terms t of our domain.

Congruence rules

$$
\frac{N_\mathsf{z} \Longrightarrow N'_\mathsf{z}}{\mathsf{rec}^A\ t\ \mathsf{with}\ \mathsf{f}\ \mathsf{z} \to N_\mathsf{z} \mid \mathsf{f}\ (\mathsf{suc}\,n) \to N_\mathsf{suc} \Longrightarrow \mathsf{rec}^A\ t\ \mathsf{with}\ \mathsf{f}\ \mathsf{z} \to N'_\mathsf{z} \mid \mathsf{f}\ (\mathsf{suc}\,n) \to N_\mathsf{suc}}
$$

$$
\frac{N_\mathsf{suc} \Longrightarrow N'_\mathsf{suc}}{\mathsf{rec}^A\ t\ \mathsf{with}\ \mathsf{f}\ \mathsf{z} \to N_\mathsf{z} \mid \mathsf{f}\ (\mathsf{suc}\,n) \to N_\mathsf{suc} \Longrightarrow \mathsf{rec}^A\ t\ \mathsf{with}\ \mathsf{f}\ \mathsf{z} \to N_\mathsf{z} \mid \mathsf{f}\ (\mathsf{suc}\,n) \to N'_\mathsf{suc}}
$$

Proving subject reduction We also revisit subject reduction, showing that the additional rules for recursion preserve types. This is a good check that we didn't screw up.

Theorem 6.1.1. *If* $M \Longrightarrow M'$ *and* $\Gamma \vdash M : C$ *then* $\Gamma \vdash M' : C$.

Proof. By structural induction on $M \Longrightarrow M'$.

Case $\mathcal{D} = \mathsf{rec}^{A}\ (\mathsf{suc}\, t)\ \mathsf{with}\ \mathsf{f}\ \mathsf{z} \to M_{\mathsf{z}} \mid \mathsf{f}\ (\mathsf{suc}\, n) \to M_{\mathsf{suc}} \implies [t/n][r/\mathsf{f}\, n]M_{\mathsf{suc}}$
where $r = \mathsf{rec}^{A}\ t\ \mathsf{with}\ \mathsf{f}\ \mathsf{z} \to M_{\mathsf{z}} \mid \mathsf{f}\ (\mathsf{suc}\, n) \to M_{\mathsf{suc}}$

$\Gamma \vdash \mathsf{rec}^{A}\ (\mathsf{suc}\, t)\mathsf{with}\ \mathsf{f}\ \mathsf{z} \to M_{\mathsf{z}} \mid \mathsf{f}\ (\mathsf{suc}\, n) \to M_{\mathsf{suc}} : C$ by assumption
$\Gamma \vdash \mathsf{suc}\, t : \mathsf{nat}$
$\Gamma \vdash M_{\mathsf{z}} : A(\mathsf{z})$
$\Gamma,\ n{:}\mathsf{nat}, \mathsf{f}\, n : A(n) \vdash M_{\mathsf{suc}} : A(\mathsf{suc}\, n)$ where $C = A(\mathsf{suc}\, t)$ by inversion on natE
$\Gamma \vdash t : \mathsf{nat}$ by $\mathsf{natI}_{\mathsf{suc}}$
$\Gamma \vdash \mathsf{rec}^{A}\ t\mathsf{with}\ \mathsf{f}\ \mathsf{z} \to M_{\mathsf{z}} \mid \mathsf{f}\ (\mathsf{suc}\, n) \to M_{\mathsf{suc}} : A(t)$ by rule natE
$[t/n][r/\mathsf{f}\, n]M_{\mathsf{suc}} : A(\mathsf{suc}\, t)$ by substitution lemma (twice)

Case $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ N_{\mathsf{z}} \Longrightarrow N'_{\mathsf{z}}\end{array}}{\mathsf{rec}^{A}\ t\ \mathsf{with}\ \mathsf{f}\ \mathsf{z} \to N_{\mathsf{z}} \mid \mathsf{f}\ (\mathsf{suc}\, n) \to N_{\mathsf{suc}} \Longrightarrow \mathsf{rec}^{A}\ t\ \mathsf{with}\ \mathsf{f}\ \mathsf{z} \to N'_{\mathsf{z}} \mid \mathsf{f}\ (\mathsf{suc}\, n) \to N_{\mathsf{suc}}}$

$\Gamma \vdash \mathsf{rec}^{A}\ t\ \mathsf{with}\ \mathsf{f}\ \mathsf{z} \to N_{\mathsf{z}} \mid \mathsf{f}\ (\mathsf{suc}\, n) \to N_{\mathsf{suc}} : C$ by assumption
$\Gamma \vdash t : \mathsf{nat}$
$\Gamma,\ n{:}\mathsf{nat}, \mathsf{f}\, n : A(n) \vdash N_{\mathsf{suc}} : A(\mathsf{suc}\, n)$
$\Gamma \vdash N_{\mathsf{z}} : A(\mathsf{z})$ where $C = A(t)$ by inversion on natE
$\Gamma \vdash N'_{\mathsf{z}} : A(\mathsf{z})$ by i.h.
$\Gamma \vdash \mathsf{rec}^{A}\ t\ \mathsf{with}\ \mathsf{f}\ \mathsf{z} \to N'_{\mathsf{z}} \mid \mathsf{f}\ (\mathsf{suc}\, n) \to N_{\mathsf{suc}} : C$ by rule natE □

6.1.4 Programming Proofs

First-order logic gives rise to dependent types and inductive proofs correspond to recursive programs. How can then program these inductive proofs? - We revisit this question using the programming and proof environment Beluga.

Beluga is a two-level system: on the first level, we encode domain specific knowledge such as the formula defining $\leq$; on the level above, we reason about such specification using first-order logic and recursion.

Let us look at the previous example where we proved that all natural numbers are less than or equal to themselves, i.e.

$$\text{For all } x\text{:nat. } x \leq x.$$

We know how to encode $\leq$ as a type family in LF as follows:

```
LF leq: nat → nat → type =
| le_z: leq z N
| le_s: leq N M → leq (s N) (s M);
```

The theorem is encoded as the type of a recursive function about the domain $\mathcal{S}$ that contains the definitions about `nat` and `leq`.

```
rec ref: {N: [⊢ nat]} [⊢ leq N N] = ? ;
```

Universal quantification is written using curly braces in Beluga. We further wrap any domain definitions we are using in brackets `[ ]`.

In the proof, we first introduced `N`. The proof term for introducing a universal quantier in Beluga is **mlam** `N` ⇒... . We then split on `N` and distinguish the case where `N` is `z` and where `N` is `s M`.

If `N = z`, then we must return a witness for `⊢ leq z z` which is `le_z`.

If `N = s M`, then by recursion (induction) we obtain a proof `P` for `⊢ leq M M`. We need to return a proof for `⊢ leq (s M) (s M)` which is described by `⊢ le_s P`.

The full Beluga program implementing the proof as a recursive program is written below:

```
rec ref: {N: [⊢ nat]} [⊢ leq N N] =
/ total n (ref n) /
mlam N ⇒ case [⊢ N] of
| [⊢ z] ⇒ [⊢ le_z]
| [⊢ s M] ⇒ let [⊢ P] = ref [⊢ M] in [⊢ le_s P];
```

We further annotate the function with `/ total n (ref n) /` which asks Beluga to check that the function `ref` is a total function which recurses over the first argument. To verify the function is total, Beluga makes sure we have covered all cases and all recursive calls are on a number that is smaller than `N`.

Remark 1 All universally quantified variables such as `N` must be capitalized. This is a syntactic convention in Beluga.

Remark 2 Any derivation we describe inside `[ ]` can only be built by either constants from our domain or variables that stand for domain objects. We cannot write `[ ⊢ le_s (ref [⊢ M])]`.