5.3.1 Substitution

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 $\mathsf{nat} : \mathsf{type}$. 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: $\mathsf{even} : \Pi n{:}\mathsf{nat}.\mathsf{type}$. The type family (predicate) lt that relates and compares two natural numbers, can be defined as: $\mathsf{lt} : \Pi n{:}\mathsf{nat}.\Pi n{:}\mathsf{nat}.\mathsf{type}$.

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 $\mathsf{a}\ M_1 \dots M_n$ where a stands for a type constant (or type family or logically a predicate) and $M_1 \dots M_n$ 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 $A \to B$.

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

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.

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

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

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

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 [**?**]. We omit the precise definition of *hereditary substitution* here, but refer the reader to for example [**?**] 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:

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

$$\mathcal{D}_1 = \boxed{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{s} \Rightarrow \mathsf{nat} \to \mathsf{nat}}\,\mathsf{const} \quad \dfrac{\dfrac{\dfrac{}{\vdash \mathsf{s} \Rightarrow \mathsf{nat} \to \mathsf{nat}}\,\mathsf{const} \quad \dfrac{\dfrac{\dfrac{}{\vdash \mathsf{s} \Rightarrow \mathsf{nat} \to \mathsf{nat}}\,\mathsf{const} \quad \dfrac{\dfrac{}{\vdash \mathsf{z} \Rightarrow \mathsf{nat}}\,\mathsf{const}}{\vdash \mathsf{z} \Leftarrow \mathsf{nat}}\,\mathsf{coe}}{\vdash \mathsf{s}\ \mathsf{z} \Rightarrow \mathsf{nat}}\,\mathsf{app}}{\vdash \mathsf{s}\ \mathsf{z} \Leftarrow \mathsf{nat}}\,\mathsf{coe}}{\vdash \mathsf{s}\ (\mathsf{s}\ \mathsf{z}) \Rightarrow \mathsf{nat}}\,\mathsf{app}}{\vdash \mathsf{s}\ (\mathsf{s}\ \mathsf{z}) \Leftarrow \mathsf{nat}}\,\mathsf{coe}}{\vdash \mathsf{s}\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{z})) \Rightarrow \mathsf{nat}}\,\mathsf{app}}{\vdash \mathsf{s}\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{z})) \Leftarrow \mathsf{nat}}\,\mathsf{coe}}$$

$$\mathcal{D}_2 = \boxed{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{ev\_s} \Rightarrow \Pi n{:}\mathsf{nat}.\mathsf{even}\ n \to \mathsf{even}\ (\mathsf{s}\ (\mathsf{s}\ n))}\,\mathsf{const} \quad \dfrac{\dfrac{}{\vdash \mathsf{z} \Rightarrow \mathsf{nat}}\,\mathsf{const}}{\vdash \mathsf{z} \Leftarrow \mathsf{nat}}\,\mathsf{coe}}{\vdash \mathsf{ev\_s}\ \mathsf{z} \Rightarrow \mathsf{even}\ \mathsf{z} \to \mathsf{even}\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{z}))}\,\mathsf{app} \quad \dfrac{\dfrac{}{\vdash \mathsf{ev\_z} \Rightarrow \mathsf{even}\ \mathsf{z}}\,\mathsf{const}}{\vdash \mathsf{ev\_z} \Leftarrow \mathsf{even}\ \mathsf{z}}\,\mathsf{coe}}{\vdash \mathsf{ev\_s}\ \mathsf{z}\ \mathsf{ev\_z} \Rightarrow \mathsf{even}\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{z}))}\,\mathsf{app}}{\vdash \mathsf{ev\_s}\ \mathsf{z}\ \mathsf{ev\_z} \Leftarrow \mathsf{even}\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{z}))}\,\mathsf{coe}}$$

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

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

As this example shows in detail, we can use the rules of the λΠ-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), **?**] 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 `ΠN: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 $\vdash \mathsf{ev\_s}\ \mathsf{z}\ \mathsf{ev\_z} \Leftarrow \mathsf{even}\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{z}))$ 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 ⊢ 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.

The following expresses a hypothetical derivation that depends on the hypothesis `u:even N` for any `N`.

```
rec hyp_d: [u: even N ⊢ even (s (s N))] =
 [u : even N ⊢ ev_s u];
```

One last remark, Beluga enforces the use of capital letters for universally quantified variables. For example, the previous statement is for all N that are natural numbers. Similarly, it forces locally bound variables, such as the label u for describing a hypothesis, to be lower case.

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)];
```

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];
```

Or more generally state that assuming that v stands for a vector of length N, we can construct a vector of length (s (s N)) for any N.

```
rec vn : [v:vec N ⊢ vec (s (s N))] = [v:vec N ⊢ snoc (snoc v tt) ff];
```

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.

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

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

$$\frac{A \text{ true} \qquad B \text{ true}}{A \wedge B \text{ true}}\wedge I \qquad \frac{A \wedge B \text{ true}}{A \text{ true}}\wedge E_l \qquad \frac{A \wedge B \text{ true}}{B \text{ true}}\wedge E_r$$

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 $\wedge E_l$ and $\wedge E_r$. The constant topI corresponds directly to the rule \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 `\u. 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 (\u. andEl u)] ;
rec p1 : [ ⊢ nd (imp (& A B) (& B A))] =
  [ ⊢ impI \u. andI (andEr u) (andEl u)];
rec p2 : [ ⊢ nd (imp (& (imp A B) (imp B C)) (imp A C))] =
  [ ⊢ impI \u. (impI \v. 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 `\u.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 (\u. fst u)] ;
rec t1 : [ ⊢ tm (imp (& A B) (& B A))] =
  [ ⊢ lam \u. pair (snd u) (fst u)];
rec t2 : [ ⊢ tm (imp (& (imp A B) (imp B C)) (imp A C))] =
  [ ⊢ lam \u. (lam \v. 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.