Proving Meta-theorems with Twelf

Dan Licata
drl@cs.cmu.edu

**********DRAFT**********
February 8th, 2006

Notes to proofreaders:

- *I haven't had time to do many of the adequacy proofs yet. If you know of a way of streamlining these, I'd love to hear it. I wonder if moving to a Kevin-style presentation would help?*
- *The theorem in Appendix A stating that equivalent worlds have the same canonical forms is true (in the sense that I wrote out the proof on paper, but I haven't had a chance to type it up yet). I haven't seen this written up anywhere, so please let me know if it exists.*

These notes are a rough draft; please alert me to any errors or omissions!

1 Introduction

1.1 Motivation

Raise your hand if you have ever seen one of these phrases in a research paper:

- "The proof proceeds by a straightforward structural induction on the first premise."
- "The remaining cases are similar."
- "For brevity, we elide the proof, which can be found in our companion technical report."

When I read one of these phrases, it makes me a little worried. How straightforward is it? Did they really do that other case? I don't usually worry enough to go and look at the tech report, though.

But, when I *write* one of these phrases, that's when I really worry. Fortunately, I don't write these phrases very often. Instead, I mechanize the meta-theory of the programming languages I design in my day-to-day research, and, in particular, I do it in Twelf [1, 16].

I'm not going to go to great lengths to sell you on mechanized metatheory here—many people have made that case (e.g., the organizers of the POPLmark Challenge [5]). I'm not even really going to try and sell you on Twelf here, except by example. This guide is intended to be a document that I can point someone towards and say, "this explains how I use Twelf in my research; give it a read and see if you think it will be useful to you." I presume no prior knowledge of LF or Twelf, though familiarity with the simply-typed λ-calculus and System F will make some of the presentation and examples more understandable.

1.2 Overview

So, what exactly is Twelf? Among other things, it includes

1. a type checker for the dependently-typed λ-calculus LF [9].
2. a meta-theorem checker.
3. an implementation of an operational semantics for LF as a logic-programming language.[1]
4. a (currently in-development) meta-theorem prover.

The first two are the focus of this guide, though I'll touch on the third as well. Since the meta-theorem prover is currently defunct (it doesn't actually produce the proof it finds), I won't discuss it at all.

Before jumping right in, it will be useful to get a big-picture sense of what's involved in using Twelf. The first step in using Twelf is encoding your language of interest in the logical framework LF. For clarity, we call the language that you wish to study the *object language* (it is the object of your study) and LF the *meta-language*. To encode a language in LF, you write an LF *signature*—a sequence of LF constant declarations—defining LF types and terms that fully and faithfully model your object language. To check that you've encoded your object language correctly, you prove a theorem called *adequacy* that establishes a bijection between the informal description of the object language (i.e., what you write on paper) and the terms of particular LF types in your signature. Once you have proven this theorem, you can reason about your object language by reasoning about its LF encoding.

LF is designed for representing deductive systems with binding—for example, programming languages and logics. Oftentimes, you will be able to represent binding in your object language with binding in LF; by doing so, you get the machinery of binding for free from the meta-language. At the level of syntax, this means that there is no need to define capture-avoiding substitution and α-equivalence for each object language you design. This technique is often called *higher-order abstract syntax*. For the judgements of your object language, you get hypothetical judgements for free by using binding in LF to represent hypotheses. This is part of the *judgements-as-types* methodology.

Once you have encoded your object language in LF, you can begin proving meta-theorems about it. First, why do we use the word "meta"-theorem? In this usage, the theorems of a deductive system are what you can prove *in* it. For example, after encoding a logic in LF you can create a derivation showing that a proposition is true; after encoding a type system, you can create a derivation showing that a program is well-typed; after encoding an operational semantics, you can create a derivation showing that a programs evaluates to a particular value. Twelf includes a "theorem checker" because it includes a type checker for LF: judgements are encoded as LF types classifying only valid derivations, so checking the validity of a derivation is the same as type checking its LF representation. On the other hand, the meta-theory of a deductive system is what you can prove *about* it. Twelf provides separate facilities for checking meta-theorems about encoded logics and languages.

1.3 Outline

The rest of this tutorial will teach you how to encode languages in LF and prove meta-theorems about them. In Section 2, you'll learn how to encode simple languages in LF. In Section 3, you'll see some first examples of meta-theorems. To keep things simple, these first object languages do not include binding. In Section 4 and Section 5 you'll learn how to encode and prove meta-theorems about languages with bindings. Since there is a lot that I won't get to cover, I am including some pointers to other examples and documentation in

[1] The original implementation of this was named Elf. You'll need a German dictionary to figure out how we got from there to here.

Section 6. Finally, I tease some advanced Twelf techniques in Section 7. Adequacy theorems and their proofs are discussed throughout, but detailed presentations of these proofs are saved for Appendix B; Appendix A presents enough of LF that we can do these proofs.

All of the examples in this document are available online [2]. They were developed and checked using the latest CVS release (Twelf 1.5R3, August 30, 2005), which I got from the anonymous CVS server:

```
cvs -d:pserver:guest_lf@cvs.concert.cs.cmu.edu:/cvsroot login
[enter blank password]
cvs -d:pserver:guest_lf@cvs.concert.cs.cmu.edu:/cvsroot checkout twelf
```

I imagine that they'd work in Twelf 1.5R1, which is available on the Twelf Web page.

2 Encoding Languages in LF: Natural Numbers with Addition

Before we can prove meta-theorems in Twelf, we need to understand how to represent the languages we'll be proving these theorems about. That is, we need to be able to represent the languages we wish to study using the logical framework LF. As a first example, adapted from Crary and Sarkar [6], let's encode a simple language of natural numbers with addition.

2.1 Natural Numbers, Informally

In informal mathematical notation, we'd write the abstract syntax of natural numbers with the following grammar:

$$\mathsf{N} \quad ::= \quad \mathsf{zero} \mid \mathsf{succ}\ \mathsf{N}.$$

The single judgement in this language, $\mathsf{N}_1 + \mathsf{N}_2 = \mathsf{N}_3$, relates two natural numbers to their sum.[2] One way of axiomatizing this relation is through the following inference rules:

$$\frac{}{\mathsf{zero} + \mathsf{N} = \mathsf{N}} \qquad \frac{\mathsf{N}_1 + \mathsf{N}_2 = \mathsf{N}_3}{\mathsf{succ}\,(\mathsf{N}_1) + \mathsf{N}_2 = \mathsf{succ}\,(\mathsf{N}_3)}.$$

Then, for example,

$$\frac{\dfrac{}{\mathsf{zero} + \mathsf{succ}\,(\mathsf{succ}\,(\mathsf{zero})) = \mathsf{succ}\,(\mathsf{succ}\,(\mathsf{zero}))}}{\mathsf{succ}\,(\mathsf{zero}) + \mathsf{succ}\,(\mathsf{succ}\,(\mathsf{zero})) = \mathsf{succ}\,(\mathsf{succ}\,(\mathsf{succ}\,(\mathsf{zero})))}$$

is a derivation of the judgement that $1 + 2 = 3$. Now that we have defined the syntax and judgements of natural numbers with addition, we can set about formalizing it. To do so, we first must define the fragment of LF that we'll use.

2.2 Simply-typed LF

LF contains the simply-typed λ-calculus. Because we will soon be working entirely in Twelf, I'm going to present LF using an abstract syntax that is close to Twelf's concrete syntax. Here's the grammar for simply-typed LF:

[2] To be precise, $\mathsf{N}_1 + \mathsf{N}_2 = \mathsf{N}_3$ is a judgement schema: there is a judgement $\mathsf{N}_1 + \mathsf{N}_2 = \mathsf{N}_3$ whenever the meta-variables N_1, N_2, and N_3 are filled in with particular natural numbers. I'm going to be sloppy about this distinction from now on.

Types	`A ::=`	`a`	base type
		`A2 -> A`	function type
Terms	`M ::=`	`c`	constant
		`x`	variable
		`[x:A] M`	λ-abstraction
		`M1 M2`	application

In its actual concrete syntax, Twelf allows any characters except `: . ( ) [ ] { } % "` and whitespace in the identifiers used for variables and constants. Parentheses are used for grouping, and the usual λ-calculus association rules apply: λ-abstractions extend as far to the right as possible, so `[x:A] M1 M2` is `[x:A] (M1 M2)` rather than `([x:A] M1) M2`; application left-associates, so `M1 M2 M3` is `(M1 M2) M3`.

LF makes a distinction between constants and variables. Type and term constants are declared in a *signature*; variables are bound by abstractions. The typing rules for the simply-typed fragment of LF are what you would expect. In addition to the typing judgement, LF also contains a notion of *definitional equality* of terms. Definitional equality is a congruent equivalence relation containing $\alpha\beta\eta$-rules. The β-normal, η-long terms are taken as *canonical forms*—the canonical representatives of the $\alpha\beta\eta$-equivalence classes of terms modulo definitional equality. We will see why term equality is important soon.

2.3 Representing Syntax in LF

Here's a rough first guideline: to represent the syntax of an object language in LF, declare base types corresponding to its syntactic categories and constants corresponding to its terms. These declarations form the LF signature that encodes your object language.

For example, we can represent the syntax of natural numbers with the following signature:

```
nat : type.
z : nat.
s : nat -> nat.
```

Notice that in the Twelf syntax, `:` is overloaded: it is used both to declare types and to declare terms of a given type; also, `.` is used to end declarations. Until you get used to thinking in LF, it can help to read these declarations out in detail. For example, the first line of the signature means "`nat` is an LF type"; the second means "`z` is an LF constant of type `nat`"; the third means "`s` is an LF constant of type `nat -> nat`".

For this particular example, it should be intuitive how these constants represent the object-language syntax. However, we should nevertheless specify this correspondence precisely by writing down the mapping from informal object-language syntax to LF terms. We'll call this the *encoding mapping*, and denote it with $\ulcorner \cdot \urcorner$. The encoding mapping is defined is defined as follows:

$$\begin{aligned} \ulcorner \mathsf{zero} \urcorner &= \mathsf{z} \\ \ulcorner \mathsf{succ}\ \mathsf{N} \urcorner &= \mathsf{s} \ulcorner \mathsf{N} \urcorner . \end{aligned}$$

2.4 Adequacy of the Encoding

Is this a good encoding? Our goal in representing a language in LF is to be able to throw away the informal description of the object language and reason using only its representation. What justifies doing so? We need a theorem that implies that any reasoning we do in or about the language as it is encoded in LF could just as well have been done in or about the informal description. In particular, this theorem would imply that when we prove meta-theoretic properties about a language's LF encoding, they are also true for the informal presentation of the language.

What should such a theorem look like? At the very least, if we say that we represent an object-language syntactic category with the elements of a particular LF type, then the encoding should be a function and its co-domain should be the desired LF type (otherwise, we haven't represented the whole object language, or we haven't represented it the way we said we would). In this case, we should check the following proposition:

PROPOSITION 2.1. *If* N *is a syntactically-correct natural number, then there exists a unique* N *such that* ⌜N⌝ = N *and* N *is an LF term of type* nat.

Proof. By induction on N. The case for zero is immediate, since ⌜zero⌝ = z and z is declared to have type nat. To show uniqueness, assume some other N' such that ⌜zero⌝ = N'; then, by inversion on the encoding, N' is z because there is only one clause that applies.

For succ N′, by induction we get that there exists a unique N' such that ⌜N′⌝ = N', where N' has type nat. Take N to be s N'; then ⌜succ N⌝ = s⌜N⌝ = s N', so such an N exists. To show uniqueness, assume some other N'' such that ⌜succ N⌝ = N''; then by inversion N'' is s⌜N⌝ because there is only one clause of the encoding that applies; this implies uniqueness because ⌜N′⌝ = N' uniquely. By the declaration in the signature we know that s has type nat -> nat, so by LF's function-application typing rule, s⌜N⌝ has type nat. □

That's a start, but it's not sufficient for throwing away the informal description of the object language and reasoning solely about its encoding in LF. For example, if we come up with an LF term of type nat, is it necessarily the encoding of some natural number N? If not, then if we prove in the formalization that there exists a nat with some properties, we don't know whether that nat is "real" according to the informal description—the formalized proof does not prove anything about the object language we have in our heads and on paper. Additionally, is there necessarily only one informal natural number gets mapped to any given LF term? If not, then when we come up with a particular LF term of type nat, we can't read back the corresponding informal object-language term.

To reason entirely in LF, the encoding mapping must be a bijection (i.e., an injective and surjective function, or a pair of mutually inverse functions); this addresses the problems in the previous paragraph. If an encoding is a bijection, then it and its inverse operation allow us to "port" any reasoning in or about the language between the informal description and its LF representation.

Unfortunately, if we read what we said literally—every LF term of type nat has a preimage under the encoding—surjectivity will never be true. It's certainly not the case that *every* LF term of type nat directly has a preimage by the encoding: that would imply that a preimage exists for any arbitrary sequence of functions and applications that winds up with type nat! However, we can rescue the situation by making use of the structure we've assumed about the space we're mapping into: we said that terms in LF are considered up to $\alpha\beta\eta$-equivalence, but we didn't mention this equivalence when we stated surjectivity. Let's try to respect equivalence:

- Surjectivity: Every LF term of type nat is equivalent to one for which a preimage exists.
- Injectivity: Any two equivalent LF terms have at most one preimage.

But what this really says is that there is a bijection between the natural numbers and the *equivalence classes* of LF terms of type nat. Observe that this is still sufficient for throwing away the informal description of the object language; for example, every LF term of type nat represents a true object-language number, namely the number that is the preimage of something in that term's equivalence class.

Rather than dealing with equivalence classes directly, we take the canonical (β-normal, η-long) forms as canonical representatives of the equivalence classes. So, to set up a bijection with the equivalence classes of LF terms of type nat, we set up a bijection with the canonical forms of type nat. This methodology

is feasible for two reasons. First, every LF term has a canonical form (up to α-equivalence), and two terms have the same canonical form iff they are in the same equivalence class. Second, the canonical (i.e., β-long, η-normal) LF terms can be characterized by an alternate, inductive definition; this simplifies proving bijections. For now, the important fact about this inductive characterization is that a constant or variable applied to canonical arguments of the correct types is canonical, as long as the iterated sequence of applications reaches base (i.e., non-function) type. For example, $\mathtt{s\ N}$ is canonical when $\mathtt{N}$ is canonical, but $\mathtt{s}$ by itself is not canonical according to this definition (indeed, nor is it η-long). Throughout, we write $\Gamma \vdash_\Sigma \mathtt{M} \overleftarrow{:} \mathtt{A}$ to denote that $\mathtt{M}$ is a canonical LF term of type $\mathtt{A}$, where the signature Σ will usually be clear from the context.

We collect the properties we require of a sensible encoding in a theorem called *adequacy*.

PROPOSITION 2.2: ADEQUACY OF NATURAL NUMBER SYNTAX. *Let Σ be the signature above. Then there is a bijection between the (informal) natural numbers as defined by the grammar and LF terms $\mathtt{N}$ such that $\cdot \vdash_\Sigma \mathtt{N} \overleftarrow{:} \mathtt{nat}$.*

We require the LF term to be well-typed in the empty context. This makes sense: the encoding we have proposed does not mention any variables, so allowing variables would break the bijection. For example, there would be no informal natural number corresponding to the LF variable $\mathtt{w}$, but $\mathtt{w} : \mathtt{nat} \vdash_\Sigma \mathtt{w} : \mathtt{nat}$.

We sketch the proof here; a more complete treatment is given in Appendix B.

Proof. First, we prove something a little stronger than Proposition 2.1, showing that $\ulcorner \cdot \urcorner$ is a function whose range is *canonical* terms of type $\mathtt{nat}$. This proof is very similar to the above, but in each case we derive canonicity rather than well-typedness. Roughly, in the first case, $\mathtt{z}$ is canonical because it is a constant of base type; in the second, we get that $\ulcorner \mathtt{N} \urcorner$ is canonical by induction, and then $\mathtt{s} \ulcorner \mathtt{N} \urcorner$ is canonical because it is a constant applied to canonical arguments and of base type.

Having established that $\ulcorner \cdot \urcorner$ is a function of the correct type, there are two easy ways to establish this bijection:

1. Define an inverse encoding $\llcorner \mathtt{N} \lrcorner$ on canonical LF terms of type $\mathtt{nat}$, show that it is a function to syntactically correct natural numbers, and then show that both compositions are the identity ($\llcorner \ulcorner \mathtt{N} \urcorner \lrcorner = \mathtt{N}$ and $\ulcorner \llcorner \mathtt{N} \lrcorner \urcorner = \mathtt{N}$).[3]

2. Show that for all canonical LF terms $\mathtt{N}$ of type $\mathtt{nat}$, there exists a unique natural number $\mathtt{N}$ such that $\ulcorner \mathtt{N} \urcorner = \mathtt{N}$. If you expand out the definitions, this is just proving injectivity and surjectivity at once.[4]

Let's do the first approach. First, you'll have to take on faith claim that, in this signature, the only canonical terms of type $\mathtt{nat}$ are $\mathtt{z}$ and $\mathtt{s\ N}$, in which case we also derive as a subderivation that $\mathtt{N}$ is canonical (or you can look at Appendix A now). Then, we define $\llcorner \mathtt{N} \lrcorner$ in the obvious way:

$$\llcorner \mathtt{z} \lrcorner = \mathsf{zero}$$

$$\llcorner \mathtt{s\ N} \lrcorner = \mathsf{succ}\left(\llcorner \mathtt{N} \lrcorner \right)$$

Now, we check properties:

- $\llcorner \cdot \lrcorner$ is a function. By induction on the derivation that $\mathtt{M}$ is canonical. It's defined for $\mathtt{z}$, and zero is syntactically correct; in the other case, we know $\mathtt{N}$ is canonical by above, so $\llcorner \mathtt{N} \lrcorner$ is defined and a correct natural number by induction, and thus $\llcorner \mathtt{s\ N} \lrcorner$ by definition exists and is correct by the grammar. Checking uniqueness is straightforward.

[3] This implies the alternate definition of a bijection as an injective and surjective function: $\llcorner \ulcorner \mathtt{N} \urcorner \lrcorner = \mathtt{N}$ implies injectivity and $\ulcorner \llcorner \mathtt{N} \lrcorner \urcorner = \mathtt{N}$ implies surjectivity.

[4] This implies that an inverse to $\ulcorner \cdot \urcorner$ exists: take $\llcorner \mathtt{N} \lrcorner$ to be the unique $\mathtt{N}$ such that $\ulcorner \mathtt{N} \urcorner = \mathtt{N}$.

- $\ulcorner \llcorner \mathtt{M} \lrcorner \urcorner = \mathtt{M}$. By induction on the derivation that $\mathtt{M}$ is canonical. It works for $\mathtt{z}$. In the other case, by induction it works for $\mathtt{N}$. Then $\llcorner \mathtt{s}\ \ \mathtt{N} \lrcorner = \mathsf{succ}\,(\llcorner \mathtt{N} \lrcorner)$, so $\ulcorner \llcorner \mathtt{s}\ \ \mathtt{N} \lrcorner \urcorner = \ulcorner \mathsf{succ}\,(\llcorner \mathtt{N} \lrcorner) \urcorner = \mathtt{s}\ \ \ulcorner \llcorner \mathtt{N} \lrcorner \urcorner$, so we get what we wanted by induction.
- $\llcorner \ulcorner \mathtt{N} \urcorner \lrcorner = \mathtt{N}$. By induction on the syntax of the natural number. It works for zero. For $\mathsf{succ}\,(\mathtt{N})$, by induction it works for $\mathtt{N}$, and then we plug through.

□

You might think that this is an awful lot of trouble to go through to establish something this simple, but keep in mind two things: first, all adequacy proofs that I've seen have gone something like this, so now you've got the model; second, adequacy is a formal way of ensuring that you're representing the language you think you're representing, which is the only way to know that all work you do in Twelf actually means anything at all.

2.5 Representing Judgements in LF, Take 1

Now that we've represented the syntax, we need to represent the $\mathtt{N}_1 + \mathtt{N}_2 = \mathtt{N}_3$ judgement. By analogy with our treatment of the syntax, let's postulate that we represent the judgement $\mathtt{N}_1 + \mathtt{N}_2 = \mathtt{N}_3$ with an LF type, and the derivations of that judgement as terms of that type. For example, let's say we represent the judgement with a type `sum`:

```
sum : type.
```

Now, we need to write constants with which we can encode the derivations. For example, we could try defining the constants

```
sum-z : nat -> sum.
sum-s : nat -> nat -> nat -> sum -> sum.
```

and encoding the object-language derivations as follows:

$$\ulcorner\ \dfrac{}{\mathsf{zero} + \mathtt{N} = \mathtt{N}}\ \urcorner = \mathtt{sum\text{-}z}\ \ulcorner \mathtt{N} \urcorner$$

$$\ulcorner\ \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \mathtt{N}_1 + \mathtt{N}_2 = \mathtt{N}_3\end{array}}{\mathsf{succ}\,(\mathtt{N}_1) + \mathtt{N}_2 = \mathsf{succ}\,(\mathtt{N}_3)}\ \urcorner = \mathtt{sum\text{-}s}\ \ulcorner \mathtt{N}_1 \urcorner \ulcorner \mathtt{N}_2 \urcorner \ulcorner \mathtt{N}_3 \urcorner \ulcorner \mathcal{D}_1 \urcorner.$$

Then, for example, `sum-z (s z)` is the representation of the derivation

$$\dfrac{}{\mathsf{zero} + \mathsf{succ}\,(\mathsf{zero}) = \mathsf{succ}\,(\mathsf{zero})}$$

and `sum-s z (s z) (s z) (sum-z (s z))` is the representation of the derivation

$$\dfrac{\dfrac{}{\mathsf{zero} + \mathsf{succ}\,(\mathsf{zero}) = \mathsf{succ}\,(\mathsf{zero})}}{\mathsf{succ}\,(\mathsf{zero}) + \mathsf{succ}\,(\mathsf{zero}) = \mathsf{succ}\,(\mathsf{succ}\,(\mathsf{zero}))}$$

Now, let's check adequacy:

CONJECTURE 2.3: ADEQUACY OF SUM. *There is a bijection between derivations of* $\mathtt{N}_1 + \mathtt{N}_2 = \mathtt{N}_3$ *and canonical LF terms* $\mathtt{D}$ *such that* $\cdot \vdash_\Sigma \mathtt{D} : \mathtt{sum}$.

Proof. First, we show that $\ulcorner \cdot \urcorner$ is a function to canonical terms of type `sum` by rule induction on $\mathcal{D}$. The proof is straightforward; it is analogous to the above proof for syntax.

However, we will not be able prove that the encoding is a bijection, as this theorem is not true! In particular, surjectivity fails. Consider the LF term

```
sum-s z z (s z) (sum-z z)
```

It is a canonical term of type `sum` because it is a constant applied to canonical arguments, but it is not the encoding of any actual derivation. Here's why: in the encoding definition, the only case whose result has the form `sum-s` $\ulcorner \mathsf{N}_1 \urcorner$ $\ulcorner \mathsf{N}_2 \urcorner$ $\ulcorner \mathsf{N}_3 \urcorner$ $\ulcorner \mathcal{D}_1 \urcorner$ has the property that $\mathcal{D}_1$ derives $\mathsf{N}_1 + \mathsf{N}_2 = \mathsf{N}_3$. However, in this case, but the preimage of `(sum-z z)` does not derive $\mathsf{zero} + \mathsf{zero} = \mathsf{succ}(\mathsf{zero})$. □

The cause of this problem is that the type `sum` is not precise enough, so the type of `sum-s` does not capture how $\mathcal{D}_1$ relates to the other arguments. Fortunately, we can fix this problem with dependent types.

2.6 Dependently Typed LF

In a dependently-typed language, types are allowed to contain terms. For example, rather than a type `sum` that classifies (the representations of) all derivations of the judgement $\mathsf{N}_1 + \mathsf{N}_2 = \mathsf{N}_3$, we could define a type `sum N1 N2 N3` that classifies only derivations relating those particular `N1`, `N2`, and `N3`.

The terms of dependently typed LF are the same as those of simply typed LF. However, the simple function type `A1 -> A2` is generalized to a dependent function type[5], written `{x:A1} A2`. The dependent function type is a binding form; `x` is bound in `A2`. Intuitively, the argument to a dependent function is allowed to appear free in the result type; application substitutes the argument into the body of the type. For example, if `c` is a constant of type `{x:nat} sum z x x`, the application `c (s z)` has type `sum z (s z) (s z)`. Twelf allows the traditional `A1 -> A2` notation as a synonym for a dependent function type `{x:A2} A` where `x` is not free in `A2`.

For dependent types to be useful, equality of types should respect equality of the terms embedded in them. For example, it is desirable that a term with type `sum (([x:nat] x) z) z z` also has type `sum z z z`. This is accomplished by extending the definitional equality relation for terms to a relation between two types that compares the embedded terms for $\alpha\beta\eta$-equality.

2.7 Representing Judgements in LF, Take 2

Let's see how we use dependent types to fix our `sum` judgement. First, we postulate a *type family* `sum N1 N2 N3` that is well-formed whenever all `Ni` have type `nat`; we'll see how to declare such a type in an LF signature in a little while. We call the `Ni` the *indices* of the type family. This terminology has set-theoretic origins: `sum` defines a family of types indexed by three `nat`s; there is one type in the family for each choice of indices.

Given this type family, we give more precise types to the constants representing the inference rules:

```
sum-z : {n : nat} sum z n n.
sum-s : {n1 : nat} {n2 : nat} {n3 : nat}
          sum n1 n2 n3 -> sum (s n1) n2 (s n3)
```

These constants seem like they capture the object language judgements much more precisely. We use the same encodings as before:

$$\ulcorner\ \overline{\mathsf{zero} + \mathsf{N} = \mathsf{N}}\ \urcorner = \texttt{sum-z}\ \ulcorner \mathsf{N} \urcorner$$

[5] In standard abstract syntax, this type is written $\Pi\, \texttt{x:A1}.\ \texttt{A2}$.

$$\ulcorner\ \dfrac{\begin{array}{c}\mathcal{D}_1\\ \mathsf{N}_1 + \mathsf{N}_2 = \mathsf{N}_3\end{array}}{\mathsf{succ}\,(\mathsf{N}_1) + \mathsf{N}_2 = \mathsf{succ}\,(\mathsf{N}_3)}\ \urcorner = \texttt{sum-s}\ \ulcorner\mathsf{N}_1\urcorner\ \ulcorner\mathsf{N}_2\urcorner\ \ulcorner\mathsf{N}_3\urcorner\ \ulcorner\mathcal{D}_1\urcorner$$

Now we can check adequacy:

PROPOSITION 2.4: ADEQUACY OF SUM. *There is a bijection between derivations of* $\mathsf{N}_1 + \mathsf{N}_2 = \mathsf{N}_3$ *and LF terms* $\mathtt{D}$ *such that* $\cdot \vdash_\Sigma \mathtt{D} \overset{\leftarrow}{:} \texttt{sum}\ \ulcorner\mathsf{N}_1\urcorner\ \ulcorner\mathsf{N}_2\urcorner\ \ulcorner\mathsf{N}_3\urcorner$.

Again, we sketch the proof here; a more detailed presentation is in Appendix B.

Proof. First, we show that $\ulcorner\cdot\urcorner$ is a function to canonical terms of type `sum` by structural induction on $\mathcal{D}$.

- Let $\mathcal{D}$ be $\dfrac{}{\mathsf{zero} + \mathsf{N} = \mathsf{N}}$. We know that $\ulcorner\mathsf{N}\urcorner$ is a canonical term of type `nat` by adequacy of syntax (PROPOSITION 2.2). Further, the constant `sum-z` has type `{n:nat} sum z n n` by the signature. Therefore $\ulcorner\mathcal{D}\urcorner$ = `sum-z` $\ulcorner\mathsf{N}\urcorner$ exists, and it is canonical at type `sum z` $\ulcorner\mathsf{N}\urcorner$ $\ulcorner\mathsf{N}\urcorner$ because it is a constant applied to a canonical argument of the appropriate type (note the substitution of $\ulcorner\mathsf{N}\urcorner$ for `n` in the result type of the application!), and the application reaches a base type. To show uniqueness, assume some other `N'` such that $\ulcorner\mathcal{D}\urcorner$ = `N'`; then inversion on the encoding function and the uniqueness of $\ulcorner\mathsf{N}\urcorner$ give the result.

- Let $\mathcal{D}$ be
$$\dfrac{\begin{array}{c}\mathcal{D}_1\\ \mathsf{N}_1 + \mathsf{N}_2 = \mathsf{N}_3\end{array}}{\mathsf{succ}\ \mathsf{N}_1 + \mathsf{N}_2 = \mathsf{succ}\ \mathsf{N}_3}$$
By adequacy of syntax, $\ulcorner\mathsf{N}_\mathsf{i}\urcorner$ are canonical terms of type `nat`. By induction $\ulcorner\mathcal{D}_1\urcorner$ is a canonical term of type `sum`$\ulcorner\mathsf{N}_1\urcorner\ulcorner\mathsf{N}_2\urcorner\ulcorner\mathsf{N}_3\urcorner$. By the signature, `sum-s` is a constant of type `{n1:nat} {n2:nat} {n3:nat} sum n1 n2 n3 -> sum (s n1) n2 (s n3)`. Thus $\ulcorner\mathcal{D}\urcorner$ = `sum-s` $\ulcorner\mathsf{N}_1\urcorner$ $\ulcorner\mathsf{N}_2\urcorner$ $\ulcorner\mathsf{N}_3\urcorner$ $\ulcorner\mathcal{D}_1\urcorner$ exists. It is unique by the usual inversion argument, using uniqueness of the subterms. It is canonical at type `sum (s` $\ulcorner\mathsf{N}_1\urcorner$`) `$\ulcorner\mathsf{N}_2\urcorner$` (s` $\ulcorner\mathsf{N}_3\urcorner$`)`—again, note the substitution of N_i for `ni`—because it is a constant applied to type-correct, canonical arguments down to base type. Then, rewriting using the definition of the syntax encoding ($\ulcorner\mathsf{succ}\,(\mathsf{N})\urcorner$ = `s` $\ulcorner\mathsf{N}\urcorner$) implies that this is what we need to show.

Checking the rest is left as an exercise. It's probably easier to do it in the second style mentioned above (there exists a unique preimage of every canonical term) rather than the first (defining the inverse explicitly). Further, you'll need to assume that the only canonical terms of type `sum N1 N2 N3` are

- `sum-z N`, where we derived as a subderivation that `N` is canonical and of type `nat`
- `sum-s N1 N2 N3 D`, where all the arguments are canonical and of the appropriate type by subderivations.

unless you want to go look at the rules for canonicity in Appendix A yourself. Then, the proof proceeds by induction on the canonicity derivation. □

2.8 Higher Kinds and Full LF

Our one undischarged promise is showing how to actually declare the type `sum N1 N2 N3` in a signature. For this, we generalize to full LF, which allows kinds (the "types" of type families) other than `type`:

Kinds	`K ::=`	`type`	the kind of types
		`{x:A} K`	dependent-function kind
Type Families	`A ::=`	`a`	family constant
		`{x:A2} A`	dependent-function type
		`A M`	application of a type family to a term
Terms	`M ::=`	`c`	term constant
		`x`	variable
		`[x:A] M`	λ-abstraction
		`M1 M2`	application

Kinds classify families, and type families with the particular kind `type` classify terms. In a signature, you can define the type of a term-level constant or the kind of a family-level constant. For example, we could declare `sum` as follows

```
sum : {n1 : nat} {n2 : nat} {n3 : nat} type.
```

Note how in the Twelf syntax, the dependent-function kind is written the same as the dependent-function type, except you can tell the difference because the kind-level one must ultimately end in `type` (the only base kind). Also, just as with the dependent-function type, Twelf allows you to write a dependent-function kind with an arrow when the parameters are not free in the body. So, we could also declare

```
sum : nat -> nat -> nat -> type.
```

since the `ni` don't get mentioned later on. The application form `A M` is used to apply type families of dependent-function kind to terms; for example, `sum N1 N2 N3` is the (iterated) application of the constant `sum` to three terms. Just like the term-level dependent-function application, the family-level application rule substitutes arguments for bound variables in the body of dependent-function kinds; if `A` has type `{x:A2} K` and `M` has type `A2`, then `A M` has type `[M/x] K`.

2.9 Summary: the LF Methodology

Let's quickly sum up what we've learned.

GUIDELINE 2.5: ENCODING SYNTAX. To encode the syntax of your object language, declare a type for each syntactic category and a constant for each piece of syntax. You should be able to prove a bijection between syntactic objects and canonical LF terms (in the empty LF context) of the corresponding types.

In particular, it often works if you encode a grammar entry of the form $\mathtt{S} ::= \ldots \mid \mathtt{t}\ \mathtt{S}_1\ \ldots\ \mathtt{S}_n \mid \ldots$ using a constant of type `S1 -> ... Sn -> S`, where `Si` is the type classifying the encodings of abstract syntax ranged over by the meta-variable $\mathtt{S}_i$.

GUIDELINE 2.6: ENCODING JUDGEMENTS. To encode the judgements of your object language, declare a type family for each judgement and a constant for each inference rule. If the judgement relates n things, the type family should have n parameters. You should be able to prove a bijection between valid derivations and canonical LF terms (in the empty context) of the corresponding types. This is known as called the *judgements-as-types* methodology, since we represent judgements as types that classify only their valid derivations.

In particular, it often works if you encode a judgement that we write on paper with the meta-variables $\mathtt{S}_1 \ldots \mathtt{S}_n$ using a family-level constant of kind $\ulcorner\mathtt{S}_1\urcorner$ `->` $\ldots$ $\ulcorner\mathtt{S}_n\urcorner$ `-> type` and if you encode an inference rule that we write on paper as

$$\frac{\mathcal{J}_1 \quad \ldots \quad \mathcal{J}_m}{\mathcal{J}}$$

(with free meta-variables $\mathtt{S}_1 \ldots \mathtt{S}_n$) using a constant of type

```
{s1 :  S1} ... {sn :  Sn } J1 -> ... Jm -> J
```

where `Si` is the type classifying the encodings of abstract syntax ranged over by the meta-variable $\mathtt{S}_i$ and `Ji` is the type classifying the encodings of derivations of $\mathcal{J}_i$.

3 Proving Meta-theorems in Twelf: Addition is Commutative

Now we can throw away the informal descriptions and get to the good stuff. For example, I'm now going to use "syntactic category" to mean both the on-paper syntactic category and the LF type representing the syntactic category.

Π_2 Sentences

Twelf's meta-theorem checker allows you to validate proofs of statements of the form $\forall\, \mathtt{x1} : \mathtt{A1} \ldots \forall\, \mathtt{xn} : \mathtt{An}\ \exists\, \mathtt{y1} : \mathtt{B1} \ldots \exists\, \mathtt{ym} : \mathtt{Bm}$ where `Ai` and `Bj` are all (canonical) LF types; statements of this form are often called Π_2 sentences. The quantifiers range over *canonical* LF terms of those types. While at first this might seem restrictive, it is strong enough to express much of the meta-theory of programming languages. Why is that? First, we represent our object language using canonical LF terms and types, so the restriction to canonical terms and types is no problem. Second, because we represent judgements as LF types, these quantifiers range not just over (the encodings of) syntactic categories, but also over (the encodings of) judgements. Third, many of the theorems we prove show that, under certain assumptions about some judgements being derivable, other judgements are also derivable.[6]

As an example of this third point, consider the usual statement of type preservation for the simply-typed λ-calculus. Presuming judgements $\cdot \vdash \mathsf{E} : \mathsf{T}$ (typing) and $\mathsf{E} \mapsto \mathsf{E}'$ (small-step operational semantics), we'd say

> If $\cdot \vdash \mathsf{E} : \mathsf{T}$ and $\mathsf{E} \mapsto \mathsf{E}'$ then $\cdot \vdash \mathsf{E}' : \mathsf{T}$.

When we mention an unbound meta-variable in a premise, what we really mean is

> For all expressions E and E', types T, and contexts Γ, if $\cdot \vdash \mathsf{E} : \mathsf{T}$ and $\mathsf{E} \mapsto \mathsf{E}'$, then $\cdot \vdash \mathsf{E}' : \mathsf{T}$.

Further, when we mention a judgement $\mathcal{J}$ like that, we mean it as shorthand for "there is a derivation of $\mathcal{J}$". So we're saying

> For all expressions E and E', types T, and contexts Γ, if there is a derivation of $\cdot \vdash \mathsf{E} : \mathsf{T}$ and there is a derivation of $\mathsf{E} \mapsto \mathsf{E}'$, then there exists a derivation of $\cdot \vdash \mathsf{E}' : \mathsf{T}$.

However, we can just as easily give names to the derivations (even if we never use them later on), which in this case is the only difference between an "if" and a "for all"; it also gets the "there exists" to look more like we want it to:

> For all expressions E and E', types T, and contexts Γ, for all derivations $\mathcal{D}_1$ of $\cdot \vdash \mathsf{E} : \mathsf{T}$ and $\mathcal{D}_2$ of $\mathsf{E} \mapsto \mathsf{E}'$, there exists a derivation $\mathcal{D}_3$ of $\cdot \vdash \mathsf{E}' : \mathsf{T}$.

Now we have a Π_2-sentence!

[6] The most-frequently encountered examples of proofs that cannot easily be checked with Twelf's meta-theorem checker are proofs by logical relations; these proofs require more complex quantifiers than Π_2 sentences afford.

3.1 Relations are Proofs of Π_2 Sentences

A Π_2 sentence of the form $\forall$ `x1 : A1` ... $\forall$ `xn : An` $\exists$ `y1 : B1` ... $\exists$ `ym : Bm` is connected to the $n + m$-ary relations among canonical LF terms of type `A1`, ...,`An`, `B1`, ...,`Bm`. In particular, such a relation that contains at least one entry for all possible combinations of canonical LF terms of types `A1`,...,`An` is a proof of the sentence: for any given inputs `a1`:`A1`,...,`an`:`An`, there will be an entry corresponding to those inputs that provides the `b1`:`B1`,...,`bn`:`Bn` witnessing the existentials. In other words, if we think of the universally-quantified types as the inputs to the relation and the existentially-quantified types as the outputs, a total relation from inputs to outputs is a proof of that sentence. Note that we do not enforce that the relation has exactly one output for each of the inputs, which is the only additional condition necessary for the relation to be a function.

This is fortunate, since we've already seen how to represent relations using dependent types: a type family and its inhabitants represent a relation between the family's indices, where indices are related when that element of the family is inhabited. For example, the type `sum` and its inhabitants represent a relation among three `nats`, where particular `N1`, `N2`, and `N3` are related when the type `sum N1 N2 N3` is inhabited. With the signature from Section 2, `z`, `z`, and `z` are in the relation specified by `sum` because `sum z z z` is inhabited; in contrast, `z`, `z`, and `s z` are not related because there is no canonical LF term of type `sum z z (s z)`. Thus, we can represent relations using the machinery we've already developed; Twelf's meta-theorem checker is just a mechanism for verifying that particular type families and their inhabitants represent *total* relations from their inputs to their outputs.

3.2 Meta-Theorem Statements

For example, say we want to prove that our `sum` judgement is commutative. That is,

> For all numbers N_1, N_2, and N_3, for all derivations of $\mathsf{N}_1 + \mathsf{N}_2 = \mathsf{N}_3$, there exists a derivation of $\mathsf{N}_2 + \mathsf{N}_1 = \mathsf{N}_3$.

By adequacy, it suffices to show

> For all `nats` `n1`, `n2`, and `n3`, for all canonical LF terms of type `sum n1 n2 n3`, there exists a canonical LF term of type `sum n2 n1 n3`.

Following the discussion above, this $\forall\exists$-statement can be proven by exhibiting a relation of type (loosely) `(n1:nat, n2:nat, n3:nat, d1:sum n1 n2 n3, d2:sum n2 n1 n3)` that is total from its first four entries to its fifth. This type of relation corresponds to the following type family:

```
sum-commutes : {n1:nat} {n2:nat} {n3:nat}
                 (sum n1 n2 n3) -> (sum n2 n1 n3) -> type.
```

The inhabitants of this type family define a relation, where there is an entry in the relation when `sum-commutes n1 n2 n3 d1 d3` is inhabited. However, with just this declaration, we haven't yet told Twelf which parameters are the inputs to the relation: we've lost the information about which variables are universally-quantified and which are existentially-quantified. For example. this type family also corresponds to all the following sentences: "for all `n1` and `n2`, there exists an `n3` such that `sum n1 n2 n3` is derivable and `sum n2 n1 n3` is derivable"; "for all `n1`, `n2`, and `n3`, there exists a derivation of `sum n1 n2 n3` and a derivation of `sum n2 n1 n3`". We fix this using a *mode* declaration:

```
%mode sum-commutes +N1 +N2 +N3 +D1 -D2.
```

`+` means "universally-quantified" or "input to the relation" and `-` means "existentially-quantified" or "output of the relation", and the variable names here can be chosen arbitrarily.

Here's a rough guideline:

GUIDELINE 3.1: ENCODING META-THEOREM STATEMENTS. To encode the statement of a meta-theorem, declare a family-level constant whose kind specifies a relation among the subjects of the theorem, along with a mode declaration identifying which parameters of the type family are the inputs to the relation and which are the outputs of the relation.

3.3 Twelf Niceties: Term Reconstruction; `<-`

3.3.1 Implicit Arguments

Writing out all the meta-variables from the informal description as explicit arguments gets tedious. Fortunately, Twelf allows implicit arguments. If you use an identifier that starts with a lower-case letter, then Twelf assumes that it will be bound somewhere. However, if you use an unbound identifier that starts with an upper-case letter in the type/kind declaration of a constant, Twelf implicitly binds it in a at the front. So, the following would be an equivalent definition of `sum-commutes`:

```
sum-commutes : (sum N1 N2 N3) -> (sum N2 N1 N3) -> type.
```

The `{Ni:nat}` are really still there; you just can't see them. How do you apply a constant to one of its implicit argument? You don't. Twelf infers the argument based the type at which the constant is used. This process is called term reconstruction.

The mode declaration correspondingly omits the implicit arguments; we'd write

```
%mode sum-commutes +D1 -D2.
```

By default, implicit arguments are quantified according to these rules:

- If the variable occurs only in universally-quantified arguments (`+`), then it is universally quantified.
- If the variable occurs only in existentially-quantified arguments (`-`), then it is existentially quantified.
- If a variable appears in both kinds of arguments, then the universal rule takes precedence.

Note that this is usually what we mean when we use implicitly-quantified meta-variables in informal math.

We can use implicit arguments to tighten up the inference rule declarations from before as well:

```
sum-z : sum z N N.
sum-s : sum N1 N2 N3 -> sum (s N1) N2 (s N3).
```

Now, rather than applying `sum-z` to a number yourself, you can let Twelf figure out what number you meant from the context you use it in.

3.3.2 Backward Arrow

Another nice bit of Twelf syntax is that you can write an `->` backward instead. That is, we could equivalently have written

```
sum-s : sum (s N1) N2 (s N3)
          <- sum N1 N2 N3.
```

This makes it easy to see what type a constant ends up at when it is fully applied.

3.4 Proofs of Meta-Theorems

Now that we know how to (concisely) state meta-theorems, how do we prove them? We have to write the constants inhabiting the type family representing the relation and then verify that the family represents a total relation.

To prove `sum-commutes`, we'll need a couple of lemmas that are the analogues of the inference rules defining `sum` but for the right-hand, rather than left-hand, summand. First, we'll prove

```
sum-z-rh : {n : nat} sum n z n -> type.
%mode sum-z-rh +N -D.
```

This type family corresponds to the following sentence:"for all (canonical) `n` of type `nat`, there exists a (canonical) derivation of type `sum n z n`".

By adequacy, this statement is equivalent to saying, "for all N, there exists a derivation of $\mathsf{zero} + \mathsf{N} = \mathsf{zero}$". Let's look at the on-paper proof first:

Proof. By induction on N.

- Case for zero: $\mathsf{zero} + \mathsf{zero} = \mathsf{zero}$ is derivable by axiom.
- Case for $\mathsf{s}(\mathsf{N}')$: By induction, there exists a derivation $\mathcal{D}$' of $\mathsf{N}' + \mathsf{zero} = \mathsf{N}'$. Then, we apply the successor rule to this derivation as follows:

$$\frac{\begin{array}{c}\mathcal{D} \\ \mathsf{N}' + \mathsf{zero} = \mathsf{N}'\end{array}}{\mathsf{succ}(\mathsf{N}') + \mathsf{zero} = \mathsf{succ}(\mathsf{N}')}$$

□

In Twelf, the corresponding cases are:

```
case-for-z : sum-z-rh z sum-z.
case-for-s : sum-z-rh (s N) (sum-s D)
               <- sum-z-rh N (D : sum N z N).
```

Let's take a look at what's going on here. In `case-for-z`, `sum-z` is precisely the the axiom we cited in the paper proof. In `case-for-s`, we "call the theorem inductively" on `N`, producing a derivation `D`, just as we did on paper; then, we simply apply `sum-s` to it, the result of which is equivalent to the derivation we wrote out in full on paper. So, by analogy at least, this seems perfectly sensible.

Now let's check by hand that we've actually written a total relation. The type family `sum-z-rh` represents a relation where `N` and `D` are related iff `sum-z-rh N D` is inhabited. We want the canonical LF terms of this type family to represent a relation that is total on `N`—that is, we want every canonical LF term of type `nat` to be in the relation. Thus, we need to add enough constants to the signature such that for all canonical LF terms `n` of type `nat`, `sum-z-rh n D` is inhabited for some `D`. `case-for-z` is a canonical LF term and covers the case when `n` is `z`. The signature says that `case-for-s X` is a canonical LF term of type `sum-z-rh (s N') (sum-s Dsum)` as long as `X` is a canonical term of type `sum-z-rh N' Dsum`. But why do we get to assume that such an `X` exists? The premise in `case-for-s` is justified by induction *over canonical forms*. Because of the way we encoded the syntax of the language, induction over canonical forms of type `nat` in LF corresponds to informal structural induction on the syntax of `N`. When on paper we appeal to induction, in Twelf we write a constant of function type whose premise is the inductive call. That the inductive call is valid is verified by induction over canonical forms. Checking that relations are total is tedious; fortunately, Twelf's meta-theorem checker does this for us.

3.5 Checking Proofs of Meta-Theorems

Now that we've written down the cases of the theorem, we enter the following declarations:

```
%worlds () (sum-z-rh _ _).
%total N (sum-z-rh N _).
```

The `%worlds` declaration defines the form of the LF contexts for which the theorem is stated; it, like the `%mode` declaration, is another part of the theorem statement. In this case, we state the theorem only for empty worlds. The `%total` declaration checks that the cases prove the stated theorem. To validate your proof of a meta-theorem, Twelf checks the following:

1. **Mode:** This actually happens as you go, not when you check totality. That is, when you enter a declaration like `case-for-z`, in addition to checking that the declared type is well-formed, Twelf checks *mode correctness* because we gave the result type `sum-z-rh` a mode declaration.

   Mode correctness is part of what justifies that all the premises of the case are reasonable. To a rough approximation, mode correctness checks that all the premises in the type of a theorem case are themselves moded, and that all the inputs in these premises come from inputs to theorem case. For example, the constant

   ```
   bogus : sum-z-rh N D.
   ```

   is type-correct because Twelf infers the type `sum N z N` for `D`. However, this "case" presumes a derivation `D`, where Twelf has no way of knowing that this derivation `D` exists. Since Twelf can't verify that a case with this premise can ever be used to create canonical terms of type `sum N D`, it doesn't help prove the theorem; thus, Twelf rejects it. Indeed, the corresponding on-paper case would read something like "Case for any `N`: the derivation exists, so we're done".

   If we try to check this declaration, Twelf will respond

   ```
   Occurrence of variable D in output (-) argument not necessarily ground
   ```

   Ground is a word from logic programming—this error message means that Twelf isn't convinced that the output comes from the input.

   A slightly different twist on this is to make up something to pass to a recursive call (or a call to a lemma): you're not allowed to make up inputs, either.

   You might think the error in for `bogus` should be a violation of termination. If we write out `bogus` explicitly, it says

   ```
   bogus: {N : nat} {D : sum N z N} sum-z-rh N D.
   ```

   So, why doesn't Twelf treat `N` as an input to the relation `sum` and try to come up with an inhabitant of `sum N z N`? Operationally, the answer is "because it doesn't"; the rule is as follows. If you write the premise of a constant as a `{}`, then Twelf only tries to fill it in only by unification (i.e., checking whether existing constraints imply that it must be equal to something else in the term). If you write the premise with an `->`, as in `case-for-s`, Twelf tries to fill it in using the meta-theorem reasoning (i.e., checking whether the inductive call is valid). This distinction comes from the logic programming operational semantics for LF, where `->` premises are called *subgoals*; the meta-theorem checker only treats these subgoals as inductive calls. In contrast, the implicitly-quantified variables in a term (i.e., the capital-letter ones) are often called *unification variables*.

2. **Worlds:** For now, we'll only prove theorems in the empty context, so worlds checking will always succeed. We'll talk more about this later on.

3. **Termination:** When we do proofs by induction, the induction argument must be explicit. The `N` in `%total N ...` tells Twelf that the proof is by induction on the first argument to the type family.

   Twelf permits inductive calls (i.e., a constant of function type has a subgoal that is the relation currently being shown total) on any term whose canonicity was derived while deriving the canonicity of the induction argument; you can find the rules for canonicity in Appendix A. Practically, for type families in empty worlds whose canonical forms are simply constants applied to canonical subterms, this amounts to allowing induction on constructor-guarded subterms of the induction argument. Because of the way we have encoded syntax and judgements, this gives structural induction on syntax and derivations.

   For example, in the case

   ```
   case-for-s : sum-z-rh (s N') (sum-s Dsum)
                  <- sum-z-rh N' Dsum.
   ```

   observe that `N'` is a strict subterm of `s N'`. If we tried a bogus induction, as in

   ```
   bogus2 : sum-z-rh N D
              <- sum-z-rh N D.
   ```

   We'd get an error such as

   ```
   Termination violation:
    ---> (N) < (N)
   ```

   Twelf also supports mutual and lexicographic induction, but we won't get to any examples of this in this guide.

4. **Coverage:** Termination checking shows that each case of the proof is valid. However, to give a total relation, we need to know not just that each case that we have given is valid, but that we have covered all of the inputs to the relation. Coverage checking happens in three phases:

   (a) **Input Coverage.** This part checks whether a proof covers all inputs to the relation. In our example, the cases

   ```
   case-for-z : sum-z-rh z sum-z.
   case-for-s : sum-z-rh (s N') (sum-s Dsum)
                  <- sum-z-rh N' Dsum.
   ```

   do cover all of the inputs to the relation. However, if we left off the first case and tried to check totality with just `case-for-s`, we'd get an error saying that there were uncovered inputs:

   ```
   Coverage error --- missing cases:
   {X1:sum z z z} |- sum-z-rh z X1.
   ```

   This error message says that we haven't inhabited the relation for `z`, and, helpfully, tells us what the type of the output of the relation should be in this case.

(b) **Output Freeness.** Output-freeness checking ensures that we don't erroneously assume that the output of an inductive calls or call or a call to another lemma is something that it is not. Roughly, the implicitly-quantified variables appearing in the outputs of calls must all be distinct unless they are constrained by the input to the call.

As a somewhat contrived example, say we wrote `case-for-s` as follows:

```
bogus4 : sum-z-rh (s N') (sum-s Dsum)
           <- sum-z-rh N' D.
           <- sum-z-rh N' D.
```

As an LF term, this case is well-typed, well-moded, and it satisfied the declared termination order. However, if you check totality of `sum-z-rh` with this case and `case-for-z`, you'll see the error

```
Constant bogus4
Occurrence of variable D in output (-) argument not free.
```

In this case, the problem is that `sum-z-rh` is a relation, not a function, so there is no guarantee that it produces the same derivation of `sum N' z N'` in each of the two calls. Thus, it is wrong to write `D` in both places, as doing so insists that the two calls output the same term.

(c) **Output Coverage.** Output coverage is another part of checking that we don't erroneously assume something about the output of inductive calls or calls to lemmas. In particular, output coverage checking ensures that we don't mistakenly assume something about the shape of an output.

In a case like

```
case-for-s : sum-z-rh (s N') (sum-s Dsum)
               <- sum-z-rh N' Dsum.
```

the output of the inductive call is a single implicitly-quantified variable, and, if the output freeness check succeeds, it must be unconstrained. Thus, this variable covers all possible outputs of the theorem.

However, sometimes we want to pattern match the output of an inductive call or a call to a lemma. For example, say we wanted to write a `sum-z-rh` case for a double-successor—not that we do, but it illustrates the output coverage problem. The case would start like this:

```
case-for-ss : sum-z-rh (s (s N')) _
               <- sum-z-rh (s N') Dsum.
```

Now, `Dsum` has type `sum (s N') z (s N')`. Suppose that to finish this case we need to *invert* this derivation to extract the derivation of `sum N' z N'`. That is, we reason as follows: by inspection of the rules, the only rule that could have derived `Dsum` is `sum-s` (because no other rule derives a conclusion of the form `sum (s _) _ _`), and in that case, we derived along the way that `sum N' z N'`. In Twelf, we can do this inversion by pattern matching the result of the inductive call:

```
case-for-ss : sum-z-rh (s (s N')) _
               <- sum-z-rh (s N') (sum-s Dsum').
```

where `Dsum'` has type `sum N' z N'`. In this example, this reasoning is fine. However, if our inversion reasoning had been wrong —if there had been some other way of deriving the output of the call—this case would not cover this other derivation. The output coverage check ensures

that all such inversions are correct—that there is indeed only one way the output could have been produced.

When you enter a constant, Twelf checks that its type is well-formed and that it is well-moded. Twelf checks worlds when you run the `%worlds` declaration. When you run the `%total`, it first checks termination for each case, and then it does coverage checking. Each part of coverage checking happens in a separate phase, so once your proof passes the input-coverage checker, you'll get any output freeness bugs; once those are eliminated, you'll find out about any output coverage errors. Once that goes through, you have a correct proof!

3.6 Finishing `sum-commutes`

Now that we understand what Twelf does when it checks a meta-theorem and how things can go wrong, we can quickly finish up proving that `sum` commutes. We'll need another lemma, this time the analogue of `sum-s` for the right-hand side:

```
sum-s-rh : sum N1 N2 N3 -> sum N1 (s N2) (s N3) -> type.
%mode sum-s-rh +D1 -D2.

- : sum-s-rh sum-z sum-z.
- : sum-s-rh (sum-s (D : sum N1' N2 N3')) (sum-s D')
     <- sum-s-rh D (D' : sum N1' (s N2) (s N3')).

%worlds () (sum-s-rh _ _).
%total D (sum-s-rh D _).
```

Make sure that you can read the type family and mode declarations as the $\forall\exists$ statement that they represent, and that you understand the proof. Because I never refer to the constants that constitute the proof of a meta-theorem, I usually call them all `-` as I've started to do here. Earlier constants are α-renamed out of the way of later ones, so you can't refer to them, but Twelf still knows the earlier ones exist.

With these lemmas proven, we can show our result:

```
sum-commute : sum N1 N2 N3 -> sum N2 N1 N3 -> type.
%mode sum-commute +D1 -D2.

- : sum-commute (sum-z : sum z N N) D
      <- sum-z-rh N D.
- : sum-commute (sum-s D : sum (s N1) N2 (s N3)) D''
      <- sum-commute D (D' : sum N2 N1 N3)
      <- sum-s-rh D' (D'' : sum N2 (s N1) (s N3)).

%worlds () (sum-commute _ _).
%total D (sum-commute D _).
```

Let's walk through each case. In the first, we are given a derivation of `sum z N N`, so we need to show `sum N z N`. We do this by appealing to the lemma `sum-z-rh` on `N`. In the second, we are given a derivation of `sum (s N1) N2 (s N3)`, so we need to show `sum N2 (s N1) (s N3)`. By induction on the derivation `D` of `sum N1 N2 N3`, we get that `sum N2 N1 N3`, and then `sum-s-rh` applied to this gives the result.

3.7 Interactive Proving

In the previous section, I presented the finished proofs to you, explaining how Twelf checks them. However, Twelf is also useful as you are writing a proof. We'll discuss a few ways to use Twelf during proof development here by developing the proof of `sum-commutes` interactively.

3.7.1 Finding Out Which Cases are Left

First, if you give a theorem statement and try to check it without supplying any cases, as in

```
sum-commute : sum N1 N2 N3 -> sum N2 N1 N3 -> type.
%mode sum-commute +D1 -D2.

%worlds () (sum-commute _ _).
%total D (sum-commute D _).
```

Twelf will complain that you haven't given any cases:

```
Coverage error --- missing cases:
{N1:nat} {N2:nat} {N3:nat} {X1:sum N1 N2 N3} {X2:sum N2 N1 N3}
   |- sum-commute X1 X2.
```

So it's up to you to pick a first case and start filling it.

Once you get one case in, if you recheck worlds and totality, Twelf will say

```
Coverage error --- missing cases:
{N1:nat} {N2:nat} {N3:nat} {X1:sum N1 N2 N3} {X2:sum N2 (s N1) (s N3)}
   |- sum-commute (sum-s X1) X2.
```

That is, once we've given one case, Twelf guesses how we are splitting up the cases (here, with one for each derivation of `sum N1 N2 N3`), and tells us which cases we are missing. This saves you from figuring out which cases are left, and it checks that you and Twelf agree on what you have already proven.

3.7.2 Type Inference

You can also get Twelf to help you fill in an individual case. Say we start by writing

```
- : sum-commute (sum-z : sum z N N) D.
```

and try to check that. Twelf responds

```
- : {N:nat} {D:sum N z N} sum-commute sum-z D.
Occurrence of variable D in output (-) argument not necessarily ground.
```

In addition to telling us there is a mode error, Twelf did type inference, telling us the type of the term that we need to fill in—in this case, the type of `D`. Thus, you can get Twelf to specialize the theorem statement for the particular case for you. In this case, the specialized theorem statements suggests the lemma that we need; if I hadn't told you up front that we'd need `sum-z-rh`, this is how we would have discovered that we do. Now, we can fill in the call to the lemma and complete the case.

As another example, if we start with

```
- : sum-commute (sum-s D : sum (s N1) N2 (s N3)) D''.
```

Twelf does type inference, specializing the theorem statement, and reports the mode error:

```
- :
   {N1:nat} {N2:nat} {N3:nat} {D:sum N1 N2 N3} {D''':sum N2 (s N1) (s N3)}
      sum-commute (sum-s D) D'''.
Occurrence of variable D'' in output (-) argument not necessarily ground.
```

Say we didn't know how to finish the case. Rather than figuring out what we can get by induction ourselves, we could have Twelf tell us by putting in the inductive call and asking it to do type inference. Checking

```
- : sum-commute (sum-s D : sum (s N1) N2 (s N3)) D''
      <- sum-commute D D'.
```

gives the following message:

```
- :
   {N1:nat} {N2:nat} {N3:nat} {D:sum N1 N2 N3} {D':sum N2 N1 N3}
      {D'':sum N2 (s N1) (s N3)} sum-commute D D' -> sum-commute (sum-s D) D''.
Occurrence of variable D'' in output (-) argument not necessarily ground
```

Twelf reports the type of `D'`, telling us what we got by induction. The types of `D'` and `D''` suggest the lemma we need to apply to finish the proof.

3.7.3 Type Annotations

In Twelf's concrete syntax, you can put a type annotation on any term by writing `M : A`. This is useful for guiding type inference in giving you clearer error messages. Also, you can use it to name the implicit arguments to type families so you don't have to α-rename them yourself when you're staring at Twelf's output.

3.7.4 Underscores

Instead of naming every implicitly quantified parameter with a capital letter, you can elide some of the names by replacing them with an _. This is useful, for example, when a particular part of the input is not relevant to a case of the proof. For now, you can think of this as being the same as giving the variable a name that you can't refer to; there is, however, a slight difference that comes up when we talk about binding.

3.8 Lies, Damn Lies, and Logic Programming

In this whole section, I've been lying to you about what Twelf's meta-theorem checker is doing. Sort of. It *is* verifying that the constants inhabiting a type family represent a total relation from inputs to outputs. However, it is doing this in a very particular way: it is checking that, when the family is interpreted as a higher-order logic program according to Twelf's logic-programming operational semantics, the family represents a total logic program from the inputs to the outputs. This is equivalent to what I've been telling you, as the logic programming interpretation boils down to searching for inhabitants of the type family; thus, proving that the type family gives a total logic program shows that the type family is inhabited for all inputs. I prefer the relation abstraction because it means I don't have to think about higher-order logic programming.

4 Encoding a Language with Binding: System F

Now that we have a basic understanding of LF and Twelf, let's move on to some more-realistic examples. In this section, we'll see how to encode programming languages with binding in LF; as an example, we use System F, the polymorphic λ-calculus [7, 18].

4.1 Encoding the Syntax

4.1.1 Higher-Order Abstract Syntax

On paper, we'd write the syntax of System F as follows:

Types	$T ::=$	u	type variable
		$T_2 \to T$	arrow type
		$\forall u.\, T$	forall type
Terms	$E ::=$	x	term variable
		$\lambda x{:}T.\, E$	function
		$E_1\ E_2$	application
		$\Lambda u.\, E$	type function
		$E[T]$	type application

Let's try to encode this in LF. Per our above methodology from Section 2.9, we define one LF type for each syntactic class. In particular, we define the following LF types:

```
tp : type.  %% System F types
tm : type. %% System F terms
```

Next, we inhabitant these types with constants for each of the syntactic forms. For some forms, we can follow our above methodology:

```
arrow : tp -> tp -> tp.  %% function type

app : tm -> tm -> tm. %% application
tapp : tm -> tp -> tm. %% type application
```

But what do we do about binding forms and variables? One option would be to use de Bruijn indices with our type `nat` from before; then we would, for example, represent term variables and functions with the following constants:

```
var : nat -> tm.
fn : nat -> tp -> tm -> tm.
```

However, if we take this approach, then we have to manually create the machinery of binding, capture-avoiding substitution, and α-equivalence for each object language (or somehow encapsulate it in a library that we can reuse). Moreover, to do the meta-theory, we will need to develop properties of binding for each object language.

In LF, one can avoid this tedium by using *higher-order abstract syntax*. The idea of higher-order abstract syntax (HOAS) is that you represent object-language binding with meta-language binding; consequently, you get the machinery of binding (capture-avoiding substitution, α-equivalence) for the object language for free. Clearly, this only works if the object language's notion of binding is coincides with LF's (so, for example, one probably cannot use HOAS to represent broken versions of Lisp), but this is usually the case.

Concretely, when using HOAS, we represent object-language variables with meta-language variables. This means that the representation of the part of an object-language term that is in the scope of a variable will have a free meta-language variable; the term must bind these variables. Using HOAS to represent System F, we arrive at the following signature:

```
tp : type.
arrow : tp -> tp -> tp.
forall : (tp -> tp) -> tp.

tm : type.
fn : tp -> (tm -> tm) -> tm.
app : tm -> tm -> tm.
tfn : (tp -> tm) -> tm.
tapp : tm -> tp -> tm.
```

In the type of `fn`, we represent the body of the function with an LF term of type `tm -> tm`; this represents the fact that the syntactic form $\lambda x{:}A.\, E$ binds x in E. $\Lambda u.\, E$ also binds a variable, but the variable that it binds ranges over types, not terms; correspondingly, the argument to `tfn` is a function of type `tp -> tm`, which binds a type variable in the body. Finally, `forall` binds a type in a type. There are no constants for System F variables: we said that we represent object-language variables with meta-language variables, and LF variables are already LF terms.

This encoding gives us capture-avoiding substitution for free. As usual, we use $\ulcorner \cdot \urcorner$ to denote the encoding mapping from the informal object language description to LF. Then, for example, the representation of a function $\lambda x{:}T_2.\, E$ is an LF term `fn` $\ulcorner T_2 \urcorner$ `([x]` $\ulcorner E \urcorner$ `)` where x is (potentially) free in $\ulcorner E \urcorner$. We will show that the object-language capture-avoiding substitution of a term E_2 for x in E, written $\{E_2/x\}E$, is represented by the meta-language capture-avoiding substitution of $\ulcorner E_2 \urcorner$ into $\ulcorner E \urcorner$, written $[\ulcorner E_2 \urcorner / x] \ulcorner E \urcorner$.

More formally, we define the encoding mapping as follows:

$$
\begin{array}{rcl}
\ulcorner u \urcorner & = & u \\
\ulcorner T_2 \to T \urcorner & = & \texttt{arrow}\ \ulcorner T_2 \urcorner\ \ulcorner T \urcorner \\
\ulcorner \forall u.\, T \urcorner & = & \texttt{forall}\,(\texttt{[u]}\ulcorner T \urcorner) \\
 & & \\
\ulcorner x \urcorner & = & x \\
\ulcorner \lambda x{:}T.\, E \urcorner & = & \texttt{fn}\ \ulcorner T \urcorner\ (\texttt{[x]}\ulcorner E \urcorner) \\
\ulcorner E_1\ E_2 \urcorner & = & \texttt{app}\ \ulcorner E_1 \urcorner\ \ulcorner E_2 \urcorner \\
\ulcorner \Lambda u.\, E \urcorner & = & \texttt{tfn}\,(\texttt{[u]}\ulcorner E \urcorner) \\
\ulcorner E[T] \urcorner & = & \texttt{tapp}\ \ulcorner E \urcorner\ \ulcorner T \urcorner
\end{array}
$$

In the variable cases, what we are really saying is that the System F variable x is represented as the LF variable x that is spelled with the same text; note the different fonts on each side of the equals sign. This ensures that when we bind the variable in LF (in `[x] ...`) we are binding the variable that shows up in the encoding of the term in which it is bound.

4.1.2 Adequacy and World Equivalence

Now that we have given an encoding, we should check adequacy. In our previous adequacy statements, the encoding of the object language was always well-formed in the empty context. Now that we have introduced

variables, this will no longer be true; for example, the encoding of a variable $\mathtt{x}$ is not an LF term of type $\mathtt{tm}$ in the empty context. The statement of adequacy about a language with binding must include a statement about the LF contexts in which the representation is adequate:

PROPOSITION 4.1: ADEQUACY OF SYSTEM F ENCODING. *Relative to the signature Σ above,*

1. *there is a bijection between System F types with free type variables in $\mathtt{u}_1,\ldots\mathtt{u}_n$ and LF terms $\mathtt{T}$ such that $\mathtt{u}_1:\mathtt{tp}\ldots\mathtt{u}_n:\mathtt{tp} \vdash_\Sigma \mathtt{T} \overset{\leftarrow}{:} \mathtt{tp}$.*

   Moreover, this bijection is compositional in the sense that $\{\mathsf{T}_2/\mathtt{u}\}\mathsf{T}$ is $[\ulcorner\mathsf{T}_2\urcorner/\ulcorner\mathtt{u}\urcorner]\ulcorner\mathsf{T}\urcorner$.

2. *There is a bijection between System F terms with free type variables in $\mathtt{u}_1,\ldots\mathtt{u}_n$ and free term variables in $\mathtt{x}_1,\ldots\mathtt{x}_m$ and LF terms E such that $\mathtt{u}_1:\mathtt{tp}\ldots\mathtt{u}_n:\mathtt{tp},\mathtt{x}_1:\mathtt{tm}\ldots\mathtt{x}_n:\mathtt{tm} \vdash_\Sigma \mathtt{E} \overset{\leftarrow}{:} \mathtt{tm}$.*

   Moreover, this bijection is compositional in the sense that $\{\mathsf{E}_2/\mathtt{x}\}\mathsf{E}$ is $[\ulcorner\mathsf{E}_2\urcorner/\ulcorner\mathtt{x}\urcorner]\ulcorner\mathsf{E}\urcorner$ and $\{\mathsf{T}_2/\mathtt{u}\}\mathsf{E}$ is $[\ulcorner\mathsf{T}_2\urcorner/\ulcorner\mathtt{u}\urcorner]\ulcorner\mathsf{E}\urcorner$.

In the proof of the second part, we would like to use the first. For example, when considering $\mathtt{tapp}\ \ulcorner\mathsf{E}\urcorner\ulcorner\mathsf{T}\urcorner$, we would like to know that $\ulcorner\mathsf{T}\urcorner$ adequately represents T. The first part of the theorem establishes this fact for contexts of the form $\mathtt{u}_1:\mathtt{tp}\ldots\mathtt{u}_n:\mathtt{tp}$. But does this fact imply anything about the canonical forms of type $\mathtt{tp}$ in contexts of the form $\mathtt{u}_1:\mathtt{tp}\ldots\mathtt{u}_n:\mathtt{tp},\mathtt{x}_1:\mathtt{tm}\ldots\mathtt{x}_n:\mathtt{tm}$?

We say that a *world* is a set of LF contexts; in particular, we will consider worlds that are generated by regular expressions. For example, $(\mathtt{u}:\mathtt{tp})*$ describes the world of LF contexts for which we stated the first part of this adequacy theorem. Then, the more general question at hand is this: if the canonical forms of a type adequately represent some object-language entity in one world, do they necessarily adequately represent the same entity in another world? In general, the answer is no: for example, if we add another variable $\mathtt{junk}:\mathtt{tp}$ to the world, the canonical forms of type $\mathtt{tp}$ will no longer adequately represent System F types with free type variables in $\mathtt{u}_1\ldots\mathtt{u}_n$.

Is it always the case that adding any extra variables to the world breaks the adequacy of a type family? No. In the example at hand, going from the world $\mathtt{u}_1:\mathtt{tp}\ldots\mathtt{u}_n:\mathtt{tp}$ to the world $\mathtt{u}_1:\mathtt{tp}\ldots\mathtt{u}_n:\mathtt{tp},\mathtt{x}_1:\mathtt{tm}\ldots\mathtt{x}_n:\mathtt{tm}$ does not change the canonical forms of type $\mathtt{tp}$ because variables of LF type $\mathtt{tm}$ cannot appear in LF terms of type $\mathtt{tp}$. This makes sense: System F terms cannot appear in System F types. But what is the general rule?

To answer this question, we define an order on worlds, with the intention that world $\mathcal{W}_1 \leq_{\mathtt{A}} \mathcal{W}_2$ if all the canonical forms of type $\mathtt{A}$ in $\mathcal{W}_1$ are also present in $\mathcal{W}_2$. Then, two worlds are *equivalent* for a type family $\mathtt{A}$ if each is less-than the other. This definition makes use of the concept of *subordination*. In general, we say that a type family $\mathtt{A}$ is subordinate to a type family $\mathtt{B}$ if canonical forms of type $\mathtt{A}$ can appear in canonical forms of type $\mathtt{B}$. For example, $\mathtt{tp}$ is subordinate to $\mathtt{tm}$ but not vice versa. Twelf tracks the subordination relation among type families in your signature.

Two worlds that are equivalent for one type family may not be equivalent for another, as the definition depends on which other type families are subordinate to the one in question. Because equivalent worlds differ only in assumptions that are not relevant to the type family $\mathtt{A}$, it is a theorem that the adequacy of $\mathtt{A}$ is preserved in all equivalent worlds. I present these definitions more formally in Appendix A.

We can now prove PROPOSITION 4.1:

Proof. First, prove that the worlds in the two parts are equivalent; then, the proof follows the techniques from Section 2. See Appendix A. □

4.2 Encoding the Static Semantics

4.2.1 Informal Static Semantics

For the static semantics of System F, we employ the following contexts:

$$\Delta ::= \cdot \mid \Delta, \mathsf{u}\ \mathsf{type}$$
$$\Gamma ::= \cdot \mid \Gamma, \mathsf{x} : \mathsf{T}$$

A context Δ is well-formed when all variables in it are distinct. A context Γ is well-formed with respect to a context Δ when all variables are distinct and every type is well-formed according to the following judgement.

$\boxed{\Delta \vdash \mathsf{T}\ \mathsf{type}}$

$$\frac{}{\Delta, \mathsf{u}\ \mathsf{type}, \Delta' \vdash \mathsf{u}\ \mathsf{type}}\ \textsc{wf-var}$$

$$\frac{\Delta \vdash \mathsf{T}_2\ \mathsf{type} \quad \Delta \vdash \mathsf{T}\ \mathsf{type}}{\Delta \vdash \mathsf{T}_2 \to \mathsf{T}\ \mathsf{type}}\ \textsc{wf-arrow} \qquad \frac{\Delta, \mathsf{u}\ \mathsf{type} \vdash \mathsf{T}\ \mathsf{type}}{\Delta \vdash \forall\, \mathsf{u}.\, \mathsf{T}\ \mathsf{type}}\ \textsc{wf-forall}$$

Type formation amounts to checking that all type variables are bound. Then, typing is defined by the following judgement.

$\boxed{\Delta\,;\,\Gamma \vdash \mathsf{E} : \mathsf{T}}$

$$\frac{}{\Delta\,;\,\Gamma, \mathsf{x} : \mathsf{T}, \Gamma' \vdash \mathsf{x} : \mathsf{T}}\ \textsc{of-var}$$

$$\frac{\Delta \vdash \mathsf{T}_2\ \mathsf{type} \quad \Delta\,;\,\Gamma, \mathsf{x} : \mathsf{T}_2 \vdash \mathsf{E} : \mathsf{T}}{\Delta\,;\,\Gamma \vdash \lambda\, \mathsf{x}{:}\mathsf{T}_2.\, \mathsf{E} : \mathsf{T}_2 \to \mathsf{T}}\ \textsc{of-fn} \qquad \frac{\Delta\,;\,\Gamma \vdash \mathsf{E}_1 : \mathsf{T}_2 \to \mathsf{T} \quad \Delta\,;\,\Gamma \vdash \mathsf{E}_2 : \mathsf{T}_2}{\Delta\,;\,\Gamma \vdash \mathsf{E}_1\ \mathsf{E}_2 : \mathsf{T}}\ \textsc{of-app}$$

$$\frac{\Delta, \mathsf{u}\ \mathsf{type}\,;\,\Gamma \vdash \mathsf{E} : \mathsf{T}}{\Delta\,;\,\Gamma \vdash \Lambda\, \mathsf{u}.\, \mathsf{E} : \forall\, \mathsf{u}.\, \mathsf{T}}\ \textsc{of-tfn} \qquad \frac{\Delta\,;\,\Gamma \vdash \mathsf{E}_1 : \forall\, \mathsf{u}.\, \mathsf{T} \quad \Delta \vdash \mathsf{T}_2\ \mathsf{type}}{\Delta\,;\,\Gamma \vdash \mathsf{E}_1[\mathsf{T}_2] : \{\mathsf{T}_2/\mathsf{u}\}\mathsf{T}}\ \textsc{of-tapp}$$

We employ the convention that there are implicit side-conditions on binding forms ensuring that bound variables are not already bound in Δ or Γ. Then the typing rules maintain the invariant that the contexts are well-formed, so there is no need to check that the context is well-formed at the leaves (e.g., in OF-VAR). We will check in Section 5 that $\Delta\,;\,\Gamma \vdash \mathsf{E} : \mathsf{T}$ implies $\Delta \vdash \mathsf{T}\ \mathsf{type}$.

4.2.2 Encoding Type Formation

In Section 2, we saw how to encode categorical judgements in LF; however, the static semantics of System F include hypothetical judgements. Following the guidelines in Section 2.9, we might postulate that we should represent the judgement $\Delta \vdash \mathsf{T}\ \mathsf{type}$ with a type family

```
wf : ctx -> tp -> type.
```

where `ctx` is some LF type representing the encoding of a type context. While this is possible, it is not the representation of hypothetical judgements that makes the best use of LF.

Let's consider what we expect of a hypothetical judgement:

- The hypothesis rule—e.g., WF-VAR—should hold. If we have assumed a derivation of a judgement, we should be able to conclude that judgement.
- The hypothetical judgement should satisfy a substitution principle. For the example above, the desired theorem is the following: If $\Delta, \mathsf{u}\ \mathsf{type} \vdash \mathsf{T}\ \mathsf{type}$ and $\Delta \vdash \mathsf{T}_2\ \mathsf{type}$ then $\Delta \vdash \{\mathsf{T}_2/\mathsf{u}\}\mathsf{T}\ \mathsf{type}$. That is, when we substitute an actual thing for the hypothesis, the judgement still holds.

One of the observations that makes Twelf so useful is this: we can represent hypothetical judgements using the binding structure of LF. That is, we represent an object-language derivation of a hypothetical judgement as an LF term with free variables, where the variables stand for the hypotheses. Then the substitution principle for the judgement is just substitution in LF.

For example, we can encode the judgement $\Delta \vdash \mathsf{T}\ \mathsf{type}$ with the following signature:

```
%% well-formed type
wf : tp -> type.

wf-arrow : wf (arrow T2 T)
             <- wf T2
             <- wf T.
wf-forall : wf (forall ([u : tp] (T u)))
              <- ({u:tp} {du : wf u} wf (T u)).
```

The encoding of derivations is defined as follows:

$$\ulcorner \dfrac{}{\Delta, \mathsf{u}\ \mathsf{type}, \Delta' \vdash \mathsf{u}\ \mathsf{type}} \urcorner \quad = \quad \texttt{du}$$

$$\ulcorner \dfrac{\overset{\mathcal{D}_2}{\Delta \vdash \mathsf{T}_2\ \mathsf{type}} \quad \overset{\mathcal{D}}{\Delta \vdash \mathsf{T}\ \mathsf{type}}}{\Delta \vdash \mathsf{T}_2 \to \mathsf{T}\ \mathsf{type}} \urcorner \quad = \quad \texttt{wf-arrow}\ \ulcorner\mathcal{D}\urcorner\ \ulcorner\mathcal{D}_2\urcorner$$

$$\ulcorner \dfrac{\overset{\mathcal{D}}{\Delta, \mathsf{u}\ \mathsf{type} \vdash \mathsf{T}\ \mathsf{type}}}{\Delta \vdash \forall \mathsf{u}.\, \mathsf{T}\ \mathsf{type}} \urcorner \quad = \quad \texttt{wf-forall([u : tp] [du : wf u]}\ \ulcorner\mathcal{D}\urcorner\texttt{)}$$

What's going on here? Hopefully the case for `wf-arrow` is straightforward. For `wf-var`, we said that we we wanted to represent uses of hypotheses as LF variables; because there is only one well-formedness hypothesis for each object-language variable u, I've chosen as a naming convention that `du` will stand for the well-formedness hypothesis about u. The type of `wf-forall` requires some explanation. In the premise, we have a derivation $\mathcal{D}$ with free variables Δ, u that may use the hypothesis rule WF-VAR on any of these variables. Thus, $\ulcorner\mathcal{D}\urcorner$ will be an LF term with free variables including `u`, `du`, and similarly for the other variables in Δ. But u is bound in the on-paper application of WF-FORALL, so neither u nor the applications of WF-VAR for it can appear below the line. Consequently, the LF representation of WF-FORALL binds both `u` and `du`. Thus, the premise of WF-FORALL is represented as a dependent function that, given a type `u` and a derivation `du` of its well-formedness, produces a derivation that `T u` is well-formed. This is reflected in the type of WF-FORALL and in the encoding.

Some other things to note:

- Because we use the LF context to represent the object-language context, the context does not appear as an argument to the type family representing the judgement.

- I've used Twelf's syntactic niceties here to suppress the binding of syntax (see Section 3.3). In a purer syntax, the signature would look like

```
wf-arrow : {T2:tp} {T:tp}
               wf T2 -> wf T -> wf (arrow T2 T).

wf-forall : {T:tp -> tp}
               ({u:tp} {du : wf u} wf (T u))
               -> wf (forall ([u : tp] (T u))).
```

  The encoding function would then apply these constants to the syntax encodings as defined above:

$$\ulcorner \dfrac{\overset{\mathcal{D}_2}{\Delta \vdash \mathsf{T}_2\ \mathsf{type}} \quad \overset{\mathcal{D}}{\Delta \vdash \mathsf{T}\ \mathsf{type}}}{\Delta \vdash \mathsf{T}_2 \to \mathsf{T}\ \mathsf{type}} \urcorner = \texttt{wf-arrow}\ \ulcorner \mathsf{T}_2 \urcorner\ \ulcorner \mathsf{T} \urcorner\ \ulcorner \mathcal{D}_2 \urcorner\ \ulcorner \mathcal{D} \urcorner$$

$$\ulcorner \dfrac{\Delta, \mathsf{u}\ \mathsf{type} \vdash \mathsf{T}\ \mathsf{type}}{\Delta \vdash \forall \mathsf{u}.\, \mathsf{T}\ \mathsf{type}} \urcorner = \texttt{wf-forall([u:tp]}\ \ulcorner \mathsf{T} \urcorner\texttt{)([u : tp] [du : wf u]}\ \ulcorner \mathcal{D} \urcorner\texttt{)}$$

  This last line points out that my notation is slightly confusing. The constant WF-FORALL must abstract over the body of the argument to the `forall`, but there is no way in LF to abstract over a term with a free variable without internalizing that free variable as a function; thus, `T` has type `tp -> tp`. Thus, $\ulcorner \mathsf{T} \urcorner$, which is supposed to be an LF term with a free variable `u`, is not `T`, but `(T u)`. I find the mnemonic of using "T" in both places to be more helpful than hurtful, though.

- With respect to the fully explicit constants in the previous bullet, our original constants that use the backward-arrow syntax (`<-`) (while still writing subgoals in the order that we'd write them on paper) end up with their arguments swapped: contrast the order of $\mathcal{D}$ and $\mathcal{D}_2$. This is annoying until you get used to it, but you'll internalize it fairly quickly. When you get a weird type error, look for this as a possible cause.

- For WF-FORALL, where did the implicit side condition that u is not already in Δ go? This is handled by α-conversion in LF: because the premise of the rule binds an LF variable, α-conversion ensures that this variable is not already in the LF context.

We should check adequacy for this encoding; the contexts in which the encoding is adequate result from the discussion above:

PROPOSITION 4.2: ADEQUACY OF $\Delta \vdash \mathsf{T}\ \mathsf{type}$. *There is a bijection between derivations of* $\mathsf{u}_1\ \mathsf{type}, \ldots, \mathsf{u}_n\ \mathsf{type} \vdash \mathsf{T}\ \mathsf{type}$ *(where* $\mathsf{u}_1\ \mathsf{type}, \ldots, \mathsf{u}_n\ \mathsf{type}$ *is a well-formed context) and LF terms* `D` *such that* $\mathtt{u}_1 : \mathtt{tp}, \mathtt{du}_1 : \mathtt{wf}\ \mathtt{u}_1, \ldots \vdash_\Sigma \mathtt{D} \overset{\leftarrow}{:} \mathtt{wf}\ \ulcorner \mathsf{T} \urcorner$.

Proof. This proof is similar to the proof for `sum`, though here we must deal with binding. In particular, you will have to ascertain that the worlds in this theorem statement are equivalent for types and terms to the worlds in which the syntax encodings are adequate. Give it a try if you like, and then take a look at Appendix B. □

4.2.3 Encoding Typing

Encoding the typing rules is mostly straightforward now that we have hypothetical judgements under our belts. We extend the signature as follows:

```
of : tm -> tp -> type.
```

```
of-fn : of (fn T2 ([x:tm] (E x))) (arrow T2 T)
          <- wf T2
          <- ({x : tm} {dx : of x T2} of (E x) T).
of-app : of (app E1 E2) T
           <- of E1 (arrow T2 T)
           <- of E2 T2.
of-tfn : of (tfn ([u:tp] (E u))) (forall ([u:tp] T u))
           <- ({u : tp} {du : wf u} of (E u) (T u)).
of-tapp : of (tapp E T2) (T T2)
             <- of E (forall ([u] T u))
             <- wf T2.
```

Then, we use the encoding:

$$\ulcorner \overline{\Delta\,;\,\Gamma, \mathsf{x}:\mathsf{T}, \Gamma' \vdash \mathsf{x}:\mathsf{T}} \urcorner = \texttt{dx}$$

$$\ulcorner \dfrac{\Delta \vdash \mathsf{T}_2\ \mathsf{type} \quad \Delta\,;\,\Gamma, \mathsf{x}:\mathsf{T}_2 \vdash \mathsf{E}:\mathsf{T}}{\Delta\,;\,\Gamma \vdash \lambda \mathsf{x}{:}\mathsf{T}_2.\,\mathsf{E}:\mathsf{T}_2 \to \mathsf{T}} \urcorner = \texttt{of-fn ([x:tm] [dx : of x } \ulcorner \mathsf{T}_2 \urcorner \texttt{] } \ulcorner \mathcal{D} \urcorner \texttt{) } \ulcorner \mathcal{D}_2 \urcorner$$

$$\ulcorner \dfrac{\Delta\,;\,\Gamma \vdash \mathsf{E}_1:\mathsf{T}_2 \to \mathsf{T} \quad \Delta\,;\,\Gamma \vdash \mathsf{E}_2:\mathsf{T}_2}{\Delta\,;\,\Gamma \vdash \mathsf{E}_1\,\mathsf{E}_2:\mathsf{T}} \urcorner = \texttt{of-app } \ulcorner \mathcal{D}_2 \urcorner\ \ulcorner \mathsf{SD}_1 \urcorner$$

$$\ulcorner \dfrac{\Delta, \mathsf{u}\ \mathsf{type}\,;\,\Gamma \vdash \mathsf{E}:\mathsf{T}}{\Delta\,;\,\Gamma \vdash \Lambda \mathsf{u}.\,\mathsf{E}:\forall \mathsf{u}.\,\mathsf{T}} \urcorner = \texttt{of-tfn ([u : tp] [du : wf u] } \ulcorner \mathcal{D} \urcorner \texttt{)}$$

$$\ulcorner \dfrac{\Delta\,;\,\Gamma \vdash \mathsf{E}_1:\forall \mathsf{u}.\,\mathsf{T} \quad \Delta \vdash \mathsf{T}_2\ \mathsf{type}}{\Delta\,;\,\Gamma \vdash \mathsf{E}_1[\mathsf{T}_2]:\{\mathsf{T}_2/\mathsf{u}\}\mathsf{T}} \urcorner = \texttt{of-tapp } \ulcorner \mathcal{D}_2 \urcorner\ \ulcorner \mathcal{D}_1 \urcorner$$

We once again represent typing assumptions using LF assumptions: the LF variable dx corresponds to the use of the on-paper use of the rule OF-VAR for x. The constant OF-FN is analogous to WF-FORALL, but in this rule we are adding a typing assumption to the context. As before, I've left the abstraction over the meta-variables that appear in the syntax implicit. Additionally, note again that the order of the derivations flips when we use the backward-arrow <- to declare the constants corresponding to the inference rules.

Given the discussion of wf, this encoding should be mostly straightforward. However, one subtlety is the result type in OF-TAPP. On, paper the result type of this rule is $\{\mathsf{T}_2/\mathsf{u}\}\mathsf{T}$. By adequacy of syntax (PROPOSITION 4.1), this is $[\ulcorner \mathsf{T}_2 \urcorner/\mathsf{u}]\ulcorner \mathsf{T} \urcorner$. As discussed above, $\ulcorner \mathsf{T} \urcorner$ is really (T u) (where T is an LF term of type tp -> tp that is well-formed independently of the bound variable u), so T $\ulcorner \mathsf{T}_2 \urcorner$ is indeed $[\ulcorner \mathsf{T}_2 \urcorner/\mathsf{u}]\ulcorner \mathsf{T} \urcorner$. This all must be reasoned out formally in the proof of adequacy:

PROPOSITION 4.3: ADEQUACY OF $\Delta\,;\,\Gamma \vdash \mathsf{E}:\mathsf{T}$.
There is a bijection between

- *derivations of* $\mathsf{u}_1\ \mathsf{type}, \ldots\,;\,\mathsf{x}_1:\mathsf{T}_1, \ldots \vdash \mathsf{E}:\mathsf{T}$, *where* $\Delta = \mathsf{u}_1\ \mathsf{type}, \ldots, \mathsf{u}_n\ \mathsf{type}$ *is well-formed and* $\Gamma = \mathsf{x}_1:\mathsf{T}_1, \ldots, \mathsf{x}_m:\mathsf{T}_m$ *is well-formed with respect to* Δ*; and*
- *LF terms* D *such that* $\mathtt{u}_1 : \mathtt{tp}, \mathtt{du}_1 : \mathtt{wf}\ \mathtt{u}_1, \ldots, \mathtt{x}_1 : \mathtt{tm}, \mathtt{dx}_1 : \mathtt{of}\ \mathtt{x}_1\ \ulcorner \mathsf{T}_1 \urcorner, \ldots \vdash_\Sigma \mathtt{D} \overleftarrow{:} \mathtt{of}\ \ulcorner \mathsf{E} \urcorner\ \ulcorner \mathsf{T} \urcorner$

Proof. See Appendix B if you get stuck. You will need to consider world equivalence to reuse the proofs of adequacy for syntax and type formation. □

4.3 Encoding the Dynamic Semantics

4.3.1 Informal Dynamic Semantics

We'll employ a call-by-value, left-to-right dynamic semantics for closed terms, where type abstractions are considered values.

Values $V ::= \lambda x{:}T_2.\, E \mid \Lambda u.\, E$

$\boxed{E \mapsto E'}$

$$\frac{E_1 \mapsto E_1'}{E_1\ E_2 \mapsto E_1'\ E_2}\ \text{STEP-APP-1} \qquad \frac{E_2 \mapsto E_2'}{V_1\ E_2 \mapsto V_1\ E_2'}\ \text{STEP-APP-2}$$

$$\frac{}{(\lambda x{:}T_2.\, E)\ V_2 \mapsto \{V_2/x\}E}\ \text{STEP-APP-BETA}$$

$$\frac{E_1 \mapsto E_1'}{E_1[T_2] \mapsto E_1'[T_2]}\ \text{STEP-TAPP-1} \qquad \frac{}{(\Lambda u.\, E)[T_2] \mapsto \{T_2/x\}E}\ \text{STEP-TAPP-BETA}$$

4.4 Encoding the Dynamic Semantics

The dynamic semantics is, for the most part, easier to encode than the static semantics. First, the dynamic semantics is given as a categorical, rather than hypothetical, judgement, so there we don't introduce any extra binding structure here. Second, substitution in the object language is modeled as substitution in LF, so there is no need to define any machinery for substitution.

The one mildly tricky part is handling the subsyntax V of values. LF doesn't give us a way to define subtypes, which is what the subsyntax would correspond to if we wanted to interpret it literally. However, we can equivalently think of the subsyntax as defining a judgement E value over all expressions such that E value is derivable exactly when E is also produced by the grammar defining values (i.e., E is also a V). For example, on paper this judgement would be defined as follows:

$$\frac{}{\lambda x{:}T.\, E\ \text{value}}\ \text{VALUE-FN} \qquad \frac{}{\Lambda u.\, E\ \text{value}}\ \text{VALUE-TFN}.$$

In the LF encoding, we make use of such a judgement, and interpret a reference to the meta-variable V in a rule as a `tm E` with an extra premise of `value E`. We enrich the signature as follows:

```
%% values
value : tm -> type.

value-fn : value (fn A E).
value-tfn : value (tfn E).

%% operational semantics
step : tm -> tm -> type.

step-app-1 : step (app E1 E2) (app E1' E2)
               <- step E1 E1'.
step-app-2 : step (app V1 E2) (app V1 E2')
               <- value V1
```

```
                <- step E2 E2'.
step-app-beta : step (app (fn T2 ([x:tm] (E x))) E2) (E E2)
                   <- value E2.
step-tapp-1 : step (tapp E1 T) (tapp E1' T)
                 <- step E1 E1'.
step-tapp-beta : step (tapp (tfn ([u:tp] E u)) T2) (E T2).
```

To define the encoding, we make use of the following lemma:

LEMMA 4.4: ADEQUACY OF VALUE JUDGEMENT. *There exists a bijection between terms* V *with free variables in* $\mathtt{u}_1,\ldots,\mathtt{x}_1,\ldots$ *and canonical LF terms* $\mathtt{D}$ *such that* $\mathtt{u}_1:\mathtt{tp},\ldots,\mathtt{x}_1:\mathtt{tm} \vdash_\Sigma \mathtt{D}:\mathtt{value}\ \ulcorner\mathsf{V}\urcorner$.

Proof. See Appendix B. □

Then the encoding is defined as follows:

$$\ulcorner \dfrac{\begin{array}{c}\mathcal{D}_1\\ \mathsf{E}_1 \mapsto \mathsf{E}_1'\end{array}}{\mathsf{E}_1\ \mathsf{E}_2 \mapsto \mathsf{E}_1'\ \mathsf{E}_2} \urcorner \quad = \quad \mathtt{step\text{-}app\text{-}1}\ \ulcorner\mathcal{D}_1\urcorner$$

$$\ulcorner \dfrac{\begin{array}{c}\mathcal{D}_2\\ \mathsf{E}_2 \mapsto \mathsf{E}_2'\end{array}}{\mathsf{V}_1\ \mathsf{E}_2 \mapsto \mathsf{V}_1\ \mathsf{E}_2'} \urcorner \quad = \quad \mathtt{step\text{-}app\text{-}2}\ \mathtt{Dv1}\ \ulcorner\mathcal{D}_2\urcorner$$

$$\ulcorner \overline{(\lambda \mathsf{x}{:}\mathsf{T}_2.\,\mathsf{E})\ \mathsf{V}_2 \mapsto \{\mathsf{V}_2/\mathsf{x}\}\mathsf{E}} \urcorner \quad = \quad \mathtt{step\text{-}app\text{-}beta}\ \mathtt{Dv2}$$

$$\ulcorner \dfrac{\begin{array}{c}\mathcal{D}_1\\ \mathsf{E}_1 \mapsto \mathsf{E}_1'\end{array}}{\mathsf{E}_1[\mathsf{T}_2] \mapsto \mathsf{E}_1'[\mathsf{T}_2]} \urcorner \quad = \quad \mathtt{step\text{-}tapp\text{-}1}\ \ulcorner\mathcal{D}_1\urcorner$$

$$\ulcorner \overline{(\Lambda\, \mathsf{u}.\,\mathsf{E})[\mathsf{T}_2] \mapsto \{\mathsf{T}_2/\mathsf{x}\}\mathsf{E}} \urcorner \quad = \quad \mathtt{step\text{-}tapp\text{-}beta}$$

where $\mathtt{Dvi}$ is the LF term corresponding to V_i by LEMMA 4.4.

The statement of adequacy is straightforward:

PROPOSITION 4.5: ADEQUACY OF DYNAMIC SEMANTICS. *For closed expressions* E *and* E'*, there is a bijection between derivations of* $\mathsf{E} \mapsto \mathsf{E}'$ *and canonical LF terms* $\mathtt{D}$ *such that* $\cdot \vdash_\Sigma \mathtt{D}:\mathtt{step}\ulcorner\mathsf{E}\urcorner\ulcorner\mathsf{E}'\urcorner$.

Proof. See Appendix B if you get stuck. □

5 Meta-theory of Languages with Binding: Properties of System F

5.1 Preservation

Preservation is relatively straightforward. For the most part, it uses only techniques that we covered in Section 3. The type family representing the theorem statement is

```
preserv : of E T
            -> step E E'
            -> of E' T
            -> type.
%mode preserv +X1 +X2 -X3.
```

Make sure that you understand how the translation of this type family into a ∀∃-statement gives the usual definition of type preservation. Later, in the `%worlds` declaration, we will declare that we are proving this theorem for closed terms; keep this in mind as you try to understand the proof.

The proof of preservation is usually given by induction on the dynamic semantics derivation. We expect to have once case for each `step` rule. Let's start with `step-app-1`:

```
- : preserv
     (of-app (DofE2 : of E2 T2) (DofE1 : of E1 (arrow T2 T)))
     (step-app-1 (DstepE1 : step E1 E1')) 
     (of-app DofE2 DofE1')
     <- preserv DofE1 DstepE1 (DofE1' : of E1' (arrow T2 T)).
```

Some notes on this case:

- **Inversion:** In this case, we use inversion as follows: `step-app-1` concludes `step (app E1 E2) (step app E1' E2)`, so the typing premise concludes `of (app E1 E2) T`; by inspection of the typing rules, the only rule that can make this conclusion for the syntactic form `(app E1 E2)` is `of-app`, in which case we have derivations `DofE2` and `DofE1` with the types noted in the case.

  In Twelf, the inversion is written by pattern-matching the typing premise as we do here. If the inversion reasoning were incorrect (i.e., if there were another rules that could have derived the conclusion), this case itself would still type check and mode check. However, if we did not otherwise cover that other rule, we would get an input coverage error when we tried to check the theorem. Thus, we can inline the inversion lemma, rather than stating it explicitly.

- **Induction:** After the inversion, it is simple to finish off the theorem by appealing to induction on the subderivation `DstepE1` and the typing derivation for `E1` that we got from inversion.

- **Order of Arguments:** As we discussed a little in Section 4 when we were doing the encoding, the order of the arguments to, for example, `of-app` is the reverse of the left-to-right order that we wrote on paper. If you see a strange type clash during the proof, look for this as a cause. For example, if we forget that the arguments should be swapped here, we get the following error:

  ```
  Type mismatch
  Expected: of 'E2 'T2
  Inferred: of 'E1 (arrow 'T2 'T)
  Head mismatch
  Ascription did not hold
  (Index object(s) did not match)

  Type mismatch
  Expected: of X2 (arrow (arrow 'T2 'T) X1)
  Inferred: of 'E1' (arrow 'T2 'T)
  Head mismatch
  Argument type did not match function domain type
  (Index object(s) did not match)

  Type mismatch
  Expected: of (app 'E1' 'E2) 'T
  ```

```
Inferred: of (app X2 'E1) X1
Free variable clash
Argument type did not match function domain type
(Index object(s) did not match)

 3 errors found
```

On to the next case:

```
- : preserv
    (of-app Dof2 Dof1)
    (step-app-2 (Dstep2 : step E2 E2') (Dval1 : value E1))
    (of-app Dof2' Dof1)
    <- preserv Dof2 Dstep2 (Dof2' : of E2' T2).
```

This case is very similar to the previous one. Note that the type annotations on the inputs are (usually) optional; I've left them off this case because they are the same as in the previous. As it turns out, we do not need `Dval1` for this case, so we could have elided its name, using an _ instead.

Now, the case for β-reduction:

```
- : preserv
    (of-app
       (DofE2 : of E2 T2)
       (of-fn (DofE : {x} {dx : of x T2} of (E x) T) (DwfT2 : wf T2)))
    (step-app-beta (Dv2 : value E2))
    (DofE E2 DofE2).
```

Some things to note:

- **Inversion:** In this case, we do two inversion—first on the application, and then on the function. Both of these are justified by the syntactic form of the left-hand side of dynamic semantics derivation, which is `(app (fn _) _)`. When we have nested inversions like this, we just extend the pattern matching deeper.
- **Substitution:** As we've noted, we get both substitution for terms and the proof that substitution preserves typing for free by encoding the syntax and judgements as we have. In this case, there is no substitution lemma necessary; we simply apply the LF function to the appropriate arguments.

The remaining cases are similar:

```
- : preserv
    (of-tapp (DwfT2 : wf T2) (DofE1 : of E1 (forall ([u] (T u)))))
    (step-tapp-1 (DstepE1 : step E1 E1'))
    (of-tapp DwfT2 DofE1')
    <- preserv DofE1 DstepE1 (DofE1' : of E1' (forall ([u] (T u)))).

- : preserv
    (of-tapp
       (DwfT2 : wf T2)
       (of-tfn
          (DofE : {u : tp} {du: wf u} of (E u) (T u))))
    step-tapp-beta
    (DofE T2 DwfT2).
```

We now check the theorem as follows:

```
%worlds () (preserv _ _ _).
%total D (preserv _ D _).
```

Some notes on checking this theorem:

- **Termination:** In each inductive call, the dynamic semantics derivation is a subderivation. In this particular proof, the typing derivation in each call is also smaller, so the proof could also be viewed as being by induction on typing (switch the `D` and the first `_` in the `%total` declaration).
- **Worlds:** We state this theorem for the empty world. However, the empty world is *not* equivalent to the world for which typing derivations are adequate for the type family `of`. So why does this theorem statement make any sense?

  First, the empty world is a subworld of the worlds in which types, terms, type well-formedness derivations, and typing derivations are adequate (in the sense that the contexts matching the empty world are a subset of the contexts matching those worlds). This means that the canonical LF terms of these types in the empty worlds will all be the image of some informal object, but it's not necessarily the case that they cover all informal objects. Indeed, if we look at the statements of adequacy for types, terms, type well-formedness derivations, and typing derivations, we see that the empty LF context corresponds to closed expressions and empty-context derivations. Thus, this statement of the theorem does *not* cover all of the informal object language, just those programs that are well-typed in the empty context. This is intentional: on paper, we usually only state preservation for terms that are well-typed in the empty context, as we only plan on evaluating closed programs.
- **Input Coverage:** We thought of this proof as proceeding by case-analysis of the dynamic semantics derivation, employing inversion on the typing derivation. It works just as well by case-analyzing the typing derivation and inverting the dynamic semantics derivation. For example, when the typing derivation ends in `of-app`, there are three transition rules that might apply, and we have a case for each. But then why don't we need cases where the typing derivation ends in `of-fn` or `of-tfn`? If `of-fn` was used, we have as a premise a derivation of `step (fn _) _`; however, no constants inhabit this type, so the case is contradictory. The input-coverage checker automatically rules out cases where the indices to type families are contradictory.

5.2 Progress

With preservation under our belts, we can move on to progress. The statement that we usually make on paper is

> If $\cdot ; \cdot \vdash \mathsf{E} : \mathsf{T}$ then either E value or there exists an E' such that $\mathsf{E} \mapsto \mathsf{E}'$.

The first question is how we represent this in Twelf. In $\forall\exists$ statements over LF types, there is no built-in general sum construct that we can use for the "or". However, it is easy enough to define such sums on a case-by-case basis: we define a judgement representing the desired sum. In this case, we write:

```
val-or-step : tm -> type.

val-or-step-step : val-or-step E
                   <- step E E'.
val-or-step-value : val-or-step E
                    <- value E.
```

That is, `val-or-step E` is derivable when either `E` is a value or `E` can take a step.

Then the statement of progress is straightforward:

```
progress : of E T
           -> val-or-step E
           -> type.
%mode progress +X1 -X2.
```

Let's start the proof, which, since we do not have any lemmas to do on, will obviously work by case-analyzing the typing judgement. As a warm-up, the cases for functions and type functions are easy, as both are values:

```
- : progress
     (of-fn DofE _)
     (val-or-step-value value-fn).

- : progress
     (of-tfn DofE)
     (val-or-step-value value-tfn).
```

Note that I've begun to leave the types to inference and elide the names of irrelevant derivations (for example, the type well-formedness derivation that is the second argument to `of-fn`). Now that I'm used to Twelf, I find it easier to read code without these annotations. Also, note that because the syntax arguments to `value-fn` and `value-tfn` are implicit, Twelf here figures out which function we're saying is a value based on the type of the theorem.

Now for application:

```
- : progress
     (of-app (DofE2 : of E2 T2) (DofE1 : of E1 (arrow T2 T)))
     _
     <- progress DofE1 (DvsE1 : val-or-step E1)
     <- progress DofE2 (DvsE2 : val-or-step E2).
```

In this incomplete case, I've left an `_` for the missing result derivation. We get started by observing that, by induction, `val-or-step E1` and `val-or-step E2`. To finish off the case, we'll have to case-analyze these derivations of `val-or-step`:

1. when `E1` takes a step, we can apply `step-app-1`;
2. when `E1` is a value and `E2` takes a step, we can apply `step-app-2`;
3. when they are both values, by a canonical forms lemma, `E1` must have the form `(fn ...)` and then we can apply `step-app-beta`.

How do we do this in Twelf? You might think that we could do the case-analysis by pattern-matching, for example by writing out the first case as follows:

```
-1 : progress
     (of-app (DofE2 : of E2 T2) (DofE1 : of E1 (arrow T2 T)))
     (step-app-1 DstE1)
     <- progress DofE1 ((val-or-step-step DstE1) : val-or-step E1)
     <- progress DofE2 (DvsE2 : val-or-step E2).
```

Then we would write two other constants (`-2` and `-3`) corresponding to the other two cases. Unfortunately, this turns out not to work. The problem is that Twelf checks output coverage on a case-by-case basis, not considering the other constants inhabiting the same type family. Thus, it is not enough that `-1`, `-2`, and `-3` together cover the output; each one must cover it individually.

However, Twelf obviously allows *input* coverage to be split across cases, so we can get around this problem by moving the case-analysis of the output into case-analysis of the input to a lemma. This is called *factoring*. In this example, we can start with a lemma

```
progress/app :  val-or-step E1
                -> val-or-step E2
                -> val-or-step (app E1 E2)
                -> type.
%mode progress/app +X1 +X2 -X3.
```

that takes as input the two inductive results from progress and produces the necessary output. However, a little thought (or trying to go through the proof) reveals that this lemma is false—for example, `E1` could be `(tfn ...)`, which is a value. Thus, `progress` needs to pass the typing derivation for `E1` to the lemma as well:

```
progress/app : of E1 (arrow T2 T1)
                -> val-or-step E1
                -> val-or-step E2
                -> val-or-step (app E1 E2)
                -> type.
%mode progress/app +X1 +X2 +X3 -X4.
```

The proof of this lemma is straightforward; the cases correspond to the three mentioned above:

```
- : progress/app
     _
     (val-or-step-step DstepE1)
     _
     (val-or-step-step (step-app-1 DstepE1)).

- : progress/app
     _
     (val-or-step-value DvalE1)
     (val-or-step-step DstepE2)
     (val-or-step-step (step-app-2 DstepE2 DvalE1)).

- : progress/app
     _
     (val-or-step-value DvalE1)
     (val-or-step-value DvalE2)
     (val-or-step-step (step-app-beta DvalE2)).

%worlds () (progress/app _ _ _ _).
%total {} (progress/app _ _ _ _).
```

Some things to note:

- **Underscores:** In the first case, I've elided the name for the derivation for `E2`, as the application can make progress regardless of what `E2` does. In all cases, I've elided the name for the typing derivation, as we never need to refer to it.
- **Canonical Forms:** If we never refer to the typing derivation, how does it solve the problem? Equivalently, what happened to the canonical forms lemma that we mentioned when we thought through the case above? The answer is that, because the lemma presumes that `E1` has type `arrow T2 T`, Twelf can do the canonical forms lemma for us just by checking for conflicts between indices. That is, the only terms for which `value E1` is derivable are `(fn T' ...)` and `(tfn ...)`; because `of E1 (arrow T2 T)` is also derivable, `E1` must be `(fn T2 ...)` because that is the only one of these two syntactic forms for which the judgement is inhabited. Thus, the third case, which on the face of it seems to just assume that `E1` is of the right shape, really does cover all the cases.

Returning to the `progress` theorem, this lemma makes the `of-app` case easy:

```
- : progress
     (of-app (DofE2 : of E2 T2) (DofE1 : of E1 (arrow T2 T)))
     DvsApp
     <- progress DofE1 (DvsE1 : val-or-step E1)
     <- progress DofE2 (DvsE2 : val-or-step E2)
     <- progress/app DofE1 DvsE1 DvsE2 (DvsApp : val-or-step (app E1 E2)).
```

The case for type application is similar; we need another little factoring lemma:

```
progress/tapp : of E1 (forall T)
                  -> val-or-step E1
                  -> {T2 : tp}
                     val-or-step (tapp E1 T2)
                  -> type.
%mode progress/tapp +X1 +X2 +X3 -X4.

- : progress/tapp
     Dof
     (val-or-step-step DstepE1)
     _
     (val-or-step-step (step-tapp-1 DstepE1)).

- : progress/tapp
     Dof
     (val-or-step-value DvalE1)
     _
     (val-or-step-step step-tapp-beta).

%worlds () (progress/tapp _ _ _ _).
%total {} (progress/tapp _ _ _ _).

- : progress
     (of-tapp (_ : wf T2) DofE1)
     DvsTpp
     <- progress DofE1 DvsE1
     <- progress/tapp DofE1 DvsE1 T2 DvsTpp.
```

Note that we simply pass the argument type `T2` itself to the lemma since the lemma does not require any derivation of a fact about the type (Check yourself: what would go wrong if we left `T2` implicit?). As above, Twelf is doing this canonical forms lemma for us in checking that the cases of `progress/tapp` are sufficient.

Now that we've covered all typing derivations that are possible in the empty context, we can successfully check the theorem:

```
%worlds () (progress _ _).
%total D (progress D _).
```

5.3 Regularity and Non-empty Contexts

We now prove a theorem called *regularity* (or often *validity*), which states the subjects of judgements are well-formed. This lemma is necessary when developing more complicated type theories; here, it is simply a useful sanity check. For System F as we have specified it, the only property to check is

If Δ is well-formed, Γ is well-formed with respect to Δ, and $\Delta\,;\,\Gamma \vdash \mathsf{E} : \mathsf{T}$ then $\Delta \vdash \mathsf{T}\ \mathsf{type}$.

As we mentioned before, a type-formation context Δ is well-formed if all variables in it are distinct; a typing context is well-formed with respect to a type-formation context Δ if all variables are distinct and $\Delta \vdash \mathsf{T}\,\mathsf{type}$ for all types T in it.

Observe that, unlike all theorems we have proven so far, this theorem is stated for arbitrary contexts. How do we encode this in the LF theorem statement? This is where the `%worlds` declaration is used. The `%worlds` declaration specifies the form of the LF contexts in which the $\forall\exists$-statement specified in a theorem statement should hold. Because we represent the object-language context with the LF context, this is how we encode the context present in the informal theorem statement. Like the `%mode` declaration, the `%worlds` declaration is part of the theorem statement; we have not paid it much attention until now only because we only stated theorems about the empty context (type safety for closed programs). Below, we will see two theorem statements that differ only their worlds (i.e., they are specified by the same type family and mode declaration) where one theorem is true and the other is false. This makes sense: changing the worlds changes the canonical forms that the theorem statement quantifies over.

The type family and mode for regularity are simple enough:

```
reg : of E T -> wf T -> type.
%mode reg +X1 -X2.
```

First, it is instructive to see that we cannot prove regularity by induction on the typing derivation if stated for empty worlds (i.e., if $\cdot\,;\,\cdot \vdash \mathsf{E} : \mathsf{T}$ then $\cdot \vdash \mathsf{T}\ \mathsf{type}$). Consider the case for functions:

```
- : reg (of-fn ([x] [dx : of x T2] DofE x dx) DwfT2) (wf-arrow DwfT DwfT2)
      <- ???
```

We need to appeal to induction on the typing premise, but we are stuck: the IH can only be applied to a derivation in the empty context, but the premise appears in a non-empty context. In Twelf, this problem manifests itself as `DofE` having type `{x:tm} {dx : of x T2} of (E x) T`, so it is not even type-correct to call the theorem inductively on `DofE`.

Thus, our ultimate `%worlds` declaration will be non-empty. In the statement of adequacy in PROPOSITION 4.3, we said that we represented object language context of the form

$$\mathsf{u}_1\,\mathsf{type}, \ldots, \mathsf{u}_n\,\mathsf{type}; \mathsf{x}_1 : \mathsf{T}_1, \ldots, \mathsf{x}_m : \mathsf{T}_m$$

with LF contexts of the form

$$\mathtt{u}_1 : \mathtt{tp}, \mathtt{du}_1 : \mathtt{wf}\ \mathtt{u}_1, \ldots, \mathtt{x}_1 : \mathtt{tm}, \mathtt{dx}_1 : \mathtt{of}\ \mathtt{x}_1\ \ulcorner \mathtt{T}_1 \urcorner, \ldots$$

We now need to know how to describe LF contexts of this form to Twelf. A first try is this:

```
%block wf-block : block {u : tp} {du: wf u}.
%block of-block : some {T1 : tp} block {x : tm} {dx : of x T1}.
%worlds (of-block | wf-block) (reg _ _).
```

A block declares a sequence of bindings appearing together in the context; for example, the first block declaration declares a unit `u : tp, du : wf u`, and the second `x : tm, dx : of x T1` for some `T1`. A theorem in worlds `(wf-block)` is declared for contexts containing any number of blocks of the form specified by `wf-block`; a theorem in worlds `(of-block | wf-block)` is declared for contexts containing any number of either `of-block` or `wf-block` in any interleaving. That is, in this case, the theorem is defined for contexts matching the regular expression `(of-block | wf-block)*`. Consequently, these contexts are called *regular worlds* (the phonetic collision with regularity is coincidental).

This regular expression describes all contexts mentioned in the adequacy statement. However, we can actually give a more precise statement about the worlds described in the adequacy statement. In particular, consider the type `T1` in a typing assumption, which in the statement of adequacy is the encoding $\ulcorner \mathtt{T}_1 \urcorner$ of some $\mathtt{T}_1$ in the object-language typing context Γ. In the statement of adequacy, we assume that the typing context Γ is well-formed with respect to Δ, which means that this type `T1` is well-formed with respect to the type well-formedness assumptions in the ambient context. Thus, a better regular expression for the worlds mentioned in the adequacy statement is this:

```
%block of+wf-block : some {T1 : tp} {dT1 : wf T1} block {x : tm} {dx : of x T1}.
%worlds (of+wf-block | wf-block) (reg _ _).
```

That is, in typing blocks, we require that there be some derivation that the type in question is well-formed. This will be important later on.

Now that we know how to state a more general theorem, how do we exploit this genericity in its proof? We can make inductive calls to the theorem in any world matching the declared expression. For example, we can complete the `of-fn` case as follows:

```
- : reg (of-fn ([x : tm] [dx : of x T2] DofE x dx) DwfT2) (wf-arrow DwfT DwfT2)
     <- ({x : tm} {dx : of x T2}
           reg (DofE x dx) (DwfT : wf T)).
```

This call is world-correct because it calls the theorem in a context containing one block matching `of-block` (note that the derivation `DwfT2` satisfies the `some` condition that `T2` is well-formed). To call a theorem in an extended context, we make the premise of the case higher-order, binding the variables in the extended context. Thinking of `reg Dof Dwf` as defining a relation from `Dof` to `Dwf` based on inhabitation of the type, this constant provides an inhabitant of the `of-fn ...` part of the argument space provided that, for arbitrary variables `x` and `dx`, we can always come up with an inhabitant of `reg (DofE x dx) DwfT` for some `DwfT`. The meta-theorem checker justifies this inductive call by induction over canonical forms—in this case, the canonical forms of higher type. Because LF respects α-conversion, the variables bound in this case are automatically "fresh".

The case for the other binding form follows a similar pattern:

```
- : reg (of-tfn ([u] [du : wf u] Dof u du)) (wf-forall Dwf)
     <- ({u} {du : wf u}
           reg (Dof u du) ((Dwf : {u} {du} wf (T u)) u du)).
```

Here, we make the inductive call in a context matching `wf-block`. In the previous case, we did not allow the result derivation of `wf T` to mention the new variables in the context, `x` and `dx`. Here, we express that `Dwf` does have free variables by giving it a function type and applying it to `u` and `du` in pattern-matching the output of the inductive call. This makes sense for our object language: derivations of type formation cannot refer to terms or typing derivations, but they can refer to types and typing derivations. Below, we will discuss how Twelf verifies that we have done this correctly.

As one would expect based on the form of the rules, the other cases do not need induction in extended contexts:

```
- : reg (of-app DofE2 DofE1) DwfT
     <- reg DofE1 (wf-arrow DwfT _).

- : reg (of-tapp Dwf2 Dof1) (Dwf _ Dwf2)
     <- reg Dof1 (wf-forall (Dwf : {u : cn} {du : wf u} wf (T u))).
```

In both cases, I've inverted the output derivation, pattern-matching against the form that it must have based on the shape of the type in question. In the first case, I've left part of the output unnamed since it is unnecessary. In the second, I've elided the type annotations that would name `T2`, so I've also let Twelf figure out what type I'm applying `Dwf` to by writing an `_` in the output (though we could just as easily name it `T2` right there).

Now, we can enter the `%worlds` declaration that we discussed above:

```
%worlds (of+wf-block | wf-block) (reg _ _).
```

Note that this must come after the cases of the theorem; unlike mode, Twelf does not check worlds incrementally. In checking this declaration, Twelf verifies that all the calls to `reg` are in contexts of the appropriate form.

Finally, we can check totality

```
%total D (reg D _).
```

and get a coverage error!

```
Coverage error --- missing cases:
{X1:tp} {#of-block:{x:tm} {dx:of x X1}} {X2:wf X1} |- reg #of-block_dx X2.
```

What went wrong? Input coverage checks that the type family `reg Dof Dwf` is inhabited for all canonical LF terms of type `of E T` in the signature and worlds provided. In this case, the variable `dx` in a context block of the form `of-block` is one such LF term, but we did not give a constant inhabiting `reg dx D` for some `D`.

Translating back using adequacy, this says that we did not cover the variable case; on paper, we prove the `of-var` case by appealing to well-formedness of Γ, which implies that all types in it are well-formed. Thus, this is where we need to make use of the type formation derivation that `of+wf-block` requires to exist. Unfortunately, it is not enough just to have such a derivation; we must also give Twelf a case of `reg` that tells it how to find it (i.e., inhabit the relation represented by `reg` for `dx`). Moreover, because the only place we can mention variables is in the context, we have to put the case for the theorem in the context. In particular, we extend the typing context blocks as follows:

```
%block of+reg-block : some {T : tp} {dT : wf T}
                      block
                       {x : tm} {dx : of x T}
                       {_ : reg dx dT}.
```

This block ensures that whenever we add a term and its typing derivation, we also add a case for `reg` showing why the type in question is well-formed.

If we try to recheck the theorem in these worlds

```
%worlds (of+reg-block | wf-block) (reg _ _).
```

we get a world violation on the `of-fn` case:

```
- : reg (of-fn ([x : tm] [dx : of x T2] DofE x dx) DwfT2) (wf-arrow DwfT DwfT2)
     <- ({x : tm} {dx : of x T2}
           reg (DofE x dx) (DwfT : wf T)).
```

This is understandable, as we changed the shape of the contexts in which `reg` is valid, and this constant does not match the new shape. To fix the problem, we need to revise the constant as follows:

```
- : reg (of-fn DofE DwfT2) (wf-arrow DwfT DwfT2)
     <- ({x} {dx : of x T2}
           {_ : reg dx DwfT2}
           reg (DofE x dx) (DwfT : wf T)).
```

Note that we now use `DwfT2` in defining the context. This corresponds closely to the on-paper case for the theorem, where we would need this derivation to argue that $\Gamma, x : T_2$ is well-formed.

And that does it! Though I've developed this theorem through several false starts for pedagogical purposes, the final result is quite simple:

```
reg : of E T -> wf T -> type.
%mode reg +X1 -X2.

- : reg (of-fn DofE DwfT2) (wf-arrow DwfT DwfT2)
     <- ({x} {dx : of x T2}
           {_ : reg dx DwfT2}
           reg (DofE x dx) (DwfT : wf T)).
- : reg (of-app DofE2 DofE1) DwfT
     <- reg DofE1 (wf-arrow DwfT _).
- : reg (of-tfn Dof) (wf-forall Dwf)
     <- ({u} {du : wf u}
           reg (Dof u du) ((Dwf : {u} {du} wf (T u)) u du)).
- : reg (of-tapp Dwf2 Dof1) (Dwf _ Dwf2)
     <- reg Dof1 (wf-forall (Dwf : {u} {du : wf u} wf (T u))).

%block wf-block : block {u : tp} {du: wf u}.
%block of+reg-block : some {T : tp} {dT : wf T}
                      block
                       {x : tm} {dx : of x T}
                       {_ : reg dx dT}.
%worlds (of+reg-block | wf-block) (reg _ _).
%total D (reg D _).
```

Some notes on this theorem:

- **Adequacy:** Since our theorem is not stated for the world in which the encoding of System F is adequate, there is a danger that we haven't proven the theorem that we started talking about using on-paper notation. To check that we've proven the right theorem, we must check that the world we proved `reg` total in is equal, for `of` and `wf`, to the worlds in which these type families are adequate.

  The block `wf-block` is exactly what we derived from the statement of adequacy. The only difference between `of+wf-block`, which we justified by adequacy, and `of+reg-block` is the additional `reg` cases; we must check that these do not alter the canonical forms. Fortunately, they do not, as `reg` is not subordinate to any of the type families in question.

- **Termination:** The only non-obvious part is why the inductive calls in the extended contexts are justified. As we said above, the meta-theorem checker justifies these by induction over canonical forms. For the canonical forms of function type, this allows you to extend the context with fresh variables and then call a theorem inductively on a higher-order subterm of the input applied to these variables. You can see the rules for canonical forms in Appendix A.

  Up until now, we've been thinking that inductive calls are justified if the induction argument is a strict subterm of the input. Induction on the canonical forms of higher type diverges from this mental model slightly: calling `DofE x dx` a subexpression of the input is suspicious, as we're applying a subterm of the input to other terms! Intuitively, however, these other expressions are variables, and α-renaming does not change the size of a term. If we had substituted non-variables, we would get a termination error.

- **Input Coverage:** Straightforward, given that the case in the context covers the typing derivation bound in the context.

- **Output Freeness and Coverage:** Freeness is straightforward. However, to check output coverage, we must check that the outputs from calls in extended contexts do not incorrectly ignore some bound variables. It is only permissible to assume that bound variables do not occur in an output when Twelf can verify that terms of the variable's type can never appear in terms of the output's type. For example, in the `of-fn` case, terms of type `tm` or `of x T` can never appear in terms of type `wf T`. In contrast, in the `of-tfn` case, terms of type `tp` and `wf u` can appear in terms of type `wf (T u)`, so we must make the result dependent on these variables. If did not do so, for example as in

  ```
  - : reg (of-tfn Dof) _
       <- ({u} {du : wf u}
             reg (Dof u du) (Dwf : wf (T u))).
  ```

  it would be an output coverage error (assuming we can even write a type-correct case, which seems difficult in this instance).

  As we mentioned above, Twelf tracks when terms of one type can appear in terms of another in something called the subordination relation; you can see the subordination relation by typing `Print.subord` in the Twelf server buffer (where it displays its output to you). Note that the current printout shows only immediate dependencies; the true relation is the transitive closure of what you see. You are only allowed to form a dependent function space `{x:A} B` when terms of type `A` may appear in terms of type `B`; however, rather than making you specify this relation up front and checking it, Twelf infers the relation based on the dependent function types that you include in the signature. This is part of why world checking is not done incrementally.

  Note that an underscore is parametrized by all relevant variables, so if we had replaced `Dwf` by `_`, it would not lead to a coverage error (assuming we could complete the case without referring to it, which in this case we cannot). This is how `_` is different from an unparameterized capital-letter variable.

For the proof of this theorem, we only needed contexts that introduce one block at a time; however, it is perfectly fine to introduce any sequence of blocks that match the regular worlds. For example, if our language had existential types, the case of `reg` for their `unpack` elimination form would use both a `wf-block` and an `of-block`.

6 Related Documentation

If you'd like to read more on LF and Twelf, here are just a few of the many available resources:

- The Twelf Wiki [3] is where I intend to post examples of all the advanced Twelf techniques that I haven't covered in this tutorial. Keep an eye on it.
- The Twelf User's Guide (available from the Twelf Web page [1]) discusses the features of Twelf and includes some small examples of proving and checking meta-theorems.
- Crary and Harper have written a high-level overview of how to believe a Twelf proof [8].
- Harper, Honsell, and Plotkin's first paper on LF introduces the representation methodology [9].
- Pfenning's logical-frameworks notes discuss representation in detail, though they only touch on meta-theory [15].
- Pfenning's notes on Computation and Deduction cover, among other things, defining adequate LF representations and proving meta-theorems relationally [12]. However, the latest version of these notes predates much of the meta-theorem checker, and thus does not discuss the particularities of working with it.
- The `examples` directory of the Twelf distribution contains many examples of deductive systems and their meta-theory. These examples should, for the most part, be understandable given what you know now.
- Over the years, Frank Pfenning and his students have written many papers on formalizing meta-theory in LF. However, many of these papers predate the meta-theorem checker, and thus they present proofs in the style we have seen but do not discuss checking them. The code accompanying many of these papers is included in the Twelf distribution, and much of it seems to have been updated to use the meta-theorem checker.

  Here are some examples:

  - Michaylov and Pfenning give some of the meta-theory of MinML [11].
  - Pfenning gives a proof of the Church-Rosser theorem for the simply-typed λ-calculus [13].
  - Pfenning gives a proof of cut elimination for intuitionistic logic [14].
  - Schürmann et al. work out some of the meta-theory of F^ω [21].
- Crary and Sarkar [6] give a brief tutorial on representing natural numbers and proving `sum-commutes`, which I have fleshed out in this guide. The remainder of the paper presents some of the LF representation and meta-theory of typed assembly language, with application to proof-carrying code.
- Simmons's undergraduate thesis presents the meta-theory of a language with references [22], though it does not discuss adequate representations.
- If you are interested in the meta-theory of LF itself, you should consult Harper and Pfenning [10].

- If you are interested in the meta-theory of Twelf's meta-theorem checker, there are many papers available. For example:
  - Schürmann's thesis and related papers discuss coverage checking [20].
  - Pientka and Pfenning discuss termination checking [17].
  - Anderson and Pfenning discuss a new feature, uniqueness checking [4]. This would save you from having to prove that certain judgements' outputs are uniquely determined by their inputs when you could get Twelf to do so for you.
- The Elf bibliography, linked from the Twelf Web page [1], cites many additional related papers.

7 What's Next?

If you've made it this far, you're well on your way to becoming a Twelf wizard. You've seen how higher-order syntax and judgements make it easy to encode deductive systems with binding. You've seen how to prove that the formalized system you reason about is equivalent to the description you wrote on paper. And you've seen how to write machine-checkable proofs of meta-theorems.

However, there are a lot of Twelf techniques that I haven't yet covered. Some of these techniques rely on aspects of Twelf that I haven't discussed in this tutorial; others are just clever uses of the machinery I've presented—but uses that I didn't think of until I'd seen them once. I mention some here so you know what you have to look forward to; hopefully, there will be some documented examples of these techniques up on the Twelf Wiki soon.

- **World Subsumption:** In this tutorial, any time a theorem calls a lemma, both the theorem and the lemma are declared to have the same worlds. This doesn't have to be the case. In general, you can call a lemma declared in one world from a theorem declared in another if Twelf knows that its argument for the totality of the lemma is still valid in the world of the theorem. This is related to subordination and world equivalence. There are techniques for dealing with situations where world subsumption is insufficient, too.
- **Catch-all Cases:** In the regularity example, we saw how to put cases for a theorem in the context. This is slightly annoying, as you then have to deal with the theorem every time you make a call in an extended context. Sometimes, you can avoid putting cases in the context by writing a catch-all case that covers the variable case without explicitly mentioning it.
- **Mutual and Lexicographic Induction:** You will sometimes need to prove two mutually referential theorems at once. Alternatively, you will need to prove a theorem by lexicographic induction on more than one of its arguments. Sometimes you will need to do both at once. Twelf supports all this through fancier `%total` declarations.
- **Explicit Termination Metrics:** Sometimes, your proof will not be structurally recursive on any of its subjects; instead, it will work by some size metric on one of the inputs. You can code up the metric as a judgement relating the derivation to its size and then induct on the `nat`.
- **`%reduces` Declarations:** These allow you to track when the output of a theorem is smaller than the input. Consequently, you can make inductive calls on the output of a lemma that returns a smaller derivation. In some situations, this can save you from having to use a metric.

- **Reasoning from False:** The coverage checker rules out many contradictory cases for you, but sometimes you will need to reason from contradictory assumptions yourself. You can do this by declaring an uninhabited type `false`, proving that certain assumptions are contradictory, and then writing lemmas that conclude anything from a term of type `false`.

- **Identity Types:** When you are, say, proving that a judgement representing a function returns a unique output, you will need to define equality. The most useful thing is often an identity type, whose only inhabitant is reflexivity. For example, for equality of `tp`s:

  ```
  id : tp -> tp -> type.
  refl : id T T.
  ```

  With this definition, it is easy to show that congruence rules are admissible, so equality of subexpressions implies equality of the whole, and it is easy to show that other type families respect equality.

- **Reverse the Polarity:** When you're working with translations from one language to another, you will often stumble upon a limitation of the coverage checker. Roughly, while you can put theorem cases in the context as we did with `reg`, these cases cannot themselves have premises (this is sometimes called "third-order coverage checking"). There is a simple workaround that often works; it involves simultaneously proving an analogous theorem with a different mode, changing a $\forall$ to an $\exists$.

- **Assumptions of Different Type:** Sometimes, rather than using the kinds of higher-order encodings we have seen here, it is useful to give assumptions a different type than the main judgement. This comes up, for example, when doing the semantics of a programming language with a store—the usual judgement is that a term is well-typed in a particular context and store typing, but variable assumptions are made for all store typings.

8 Acknowledgments

Karl Crary taught me this material in his version of *Computation and Deduction* here at CMU in Fall 2004, and he has provided invaluable Twelf help thereafter. Many of the examples in this guide are derived from examples we did in class. Karl Crary, William Lovas, Susmit Sarkar, and Kevin Watkins all provided pointers to related sources of documentation. Thanks!

References

[1] `http://www.twelf.org`.

[2] `http://www.cs.cmu.edu/~drl/`.

[3] `http://fp.logosphere.cs.cmu.edu/twelf/`.

[4] P. Anderson and F. Pfenning. Verifying uniqueness in a logical framework. In *International Conference on Theorem Proving in Higher Order Logics*, 2004.

[5] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, , and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *International Conference on Theorem Proving in Higher Order Logics*, 2005.

[6] K. Crary and S. Sarkar. Foundational certified code in a metalogical framework. In *Nineteenth International Conference on Automated Deduction*, 2003.

[7] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[8] R. Harper and K. Crary. How to believe a Twelf proof. http://www.cs.cmu.edu/ rwh/, 2005.

[9] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1), 1993.

[10] R. Harper and F. Pfenning. On equivalence and canonical forms in the LF type theory. *Transactions on Computational Logic*, 2003.

[11] S. Michaylov and F. Pfenning. Natural semantics and some of its meta-theory in Elf. In *International Workshop on Extensions of Logic Programming*, 1991.

[12] F. Pfenning. Computation and deduction. Draft course notes; available from `http://www.cs.cmu.edu/~fp/`.

[13] F. Pfenning. A proof of the church-rosser theorem and its representation in a logical framework. journal. *Journal of Automated Reasoning*, 1993.

[14] F. Pfenning. A structural proof of cut elimination and its representation in a logical framework. Technical Report CMU-CS-94-218, Department of Computer Science, Carnegie Mellon University, 1994.

[15] F. Pfenning. Logical frameworks. *Handbook of Automated Reasoning*, 1999.

[16] F. Pfenning and C. Schrmann. System description: Twelf - a meta-logical framework for deductive systems. In *International Conference on Automated Deduction*, 1999.

[17] B. Pientka and F. Pfenning. Termination and reduction checking in the logical framework. In *Workshop on Automation of Proofs by Mathematical Induction*, 2000.

[18] J. C. Reynolds. Towards a theory of type structure. In *Symposium on Programming*, 1974.

[19] S. Sarkar. Metatheory of LF extended with dependent pair and unit types. Technical Report CMU-CS-05-179, Carnegie Mellon University, 2005.

[20] C. Sch urmann. Automating the meta-theory of deductive systems. Technical Report CMU-CS-00-146, Department of Computer Science, Carnegie Mellon University, 2000.

[21] C. Sch urmann, D. Yu, and Z. Ni. A representation of f_ω in LF. *Electronic Notes in Theoretical Computer Science*, 58(1), 2001.

[22] R. Simmons. Twelf as a unified framework for language formalization and implementation. Technical report, Princeton University, 2005.

A Interface to LF

In this section, I present as much of LF as is necessary for the adequacy proofs below.

A.1 Syntax

I'm now going to use a more mathematical notation for LF, rather than sticking so close to the concrete syntax. Here is the correspondence with the notation we have been using so far:

Kinds	`K ::=`	`type`	`type`	the kind of types
		`{x:A} K`	Π `x:A.K`	dependent-function kind
Type Families	`A ::=`	`a`	`a`	family constant
		`{x:A2} A`	Π `x:A2.A`	dependent-function type
		`A M`	`A M`	application of a type family to a term
Terms	`M ::=`	`c`	`c`	term constant
		`x`	`x`	variable
		`[x:A] M`	λ `x:A.M`	λ-abstraction
		`M1 M2`	`M1 M2`	application

A.2 Definition of Canonical Forms

The canonical forms of LF are defined by seven judgements:

- $\Sigma\ \overleftarrow{\texttt{sig}}$, read "$\Sigma$ is a canonical signature"
- $\vdash_\Sigma\ \Gamma\ \overleftarrow{\texttt{ctx}}$, read "$\Gamma$ is a canonical context"
- $\Gamma \vdash_\Sigma$ `K` $\overleftarrow{\texttt{kind}}$, read "`K` is a canonical kind"
- $\Gamma \vdash_\Sigma$ `A` $\overleftarrow{:}$ `K`, read "`A` is canonical at `K`"
- $\Gamma \vdash_\Sigma$ `A` $\overrightarrow{:}$ `K`, read "`A` is atomic at `K`"
- $\Gamma \vdash_\Sigma$ `M` $\overleftarrow{:}$ `A`, read "`M` is canonical at `A`"
- $\Gamma \vdash_\Sigma$ `M` $\overrightarrow{:}$ `A`, read "`M` is atomic at `A`"

We make use of three auxiliary judgements,

- $\Gamma \vdash_\Sigma$ `K` $\Rightarrow$ `K'` `kind`, read "`K` canonizes to `K'`"
- $\Gamma \vdash_\Sigma$ `A` $\Rightarrow$ `A'` : `K`, read "`A` canonizes to `A'` at kind `K`"
- $\Gamma \vdash_\Sigma$ `M` $\Rightarrow$ `M'` : `A`, read "`M` canonizes to `M'` at type `A`"

We do not define the auxiliary judgements explicitly; many treatments are available [10, 19]. We will only interact with them through the properties cited below.

The primary judgements are defined by the following rules:

$\boxed{\Sigma\ \overleftarrow{\texttt{sig}}}$

$$\frac{}{\cdot\ \overleftarrow{\texttt{sig}}}\ \text{CANON-SIG-EMPTY} \qquad \frac{\Sigma\ \overleftarrow{\texttt{sig}} \quad \cdot \vdash_\Sigma \texttt{A}\ \overleftarrow{:}\ \texttt{type}}{\Sigma, \texttt{c} : \texttt{A}\ \overleftarrow{\texttt{sig}}}\ \text{CANON-SIG-TERM}$$

$$\frac{\Sigma \overset{\leftarrow}{\mathtt{sig}} \quad \cdot \vdash_\Sigma \mathtt{K} \overset{\leftarrow}{\mathtt{kind}}}{\Sigma, \mathtt{a} : \mathtt{K} \overset{\leftarrow}{\mathtt{sig}}} \text{ CANON-SIG-FAM}$$

$\boxed{\vdash_\Sigma \Gamma \overset{\leftarrow}{\mathtt{ctx}}}$

$$\frac{}{\vdash_\Sigma \cdot \overset{\leftarrow}{\mathtt{ctx}}} \text{ CANON-CTX-EMPTY} \qquad \frac{\vdash_\Sigma \Gamma \overset{\leftarrow}{\mathtt{ctx}} \quad \Gamma \vdash_\Sigma \mathtt{A} \overset{\leftarrow}{:} \mathtt{type}}{\vdash_\Sigma \Gamma, \mathtt{x} : \mathtt{A} \overset{\leftarrow}{\mathtt{ctx}}} \text{ CANON-CTX-TERM}$$

$\boxed{\Gamma \vdash_\Sigma \mathtt{K} \overset{\leftarrow}{\mathtt{kind}}}$

$$\frac{}{\Gamma \vdash_\Sigma \mathtt{type} \overset{\leftarrow}{\mathtt{kind}}} \text{ CANON-KIND-TYPE} \qquad \frac{\Gamma, \mathtt{x} : \mathtt{A} \vdash_\Sigma \mathtt{K} \overset{\leftarrow}{\mathtt{kind}}}{\Gamma \vdash_\Sigma \Pi \mathtt{x}{:}\mathtt{A}.\, \mathtt{K} \overset{\leftarrow}{\mathtt{kind}}} \text{ CANON-KIND-PI}$$

$\boxed{\Gamma \vdash_\Sigma \mathtt{A} \overset{\leftarrow}{:} \mathtt{K}}$

$$\frac{\Gamma \vdash_\Sigma \mathtt{A}_2 \overset{\leftarrow}{:} \mathtt{type} \quad \Gamma, \mathtt{x} : \mathtt{A}_2 \vdash_\Sigma \mathtt{A} \overset{\leftarrow}{:} \mathtt{type}}{\Gamma \vdash_\Sigma \Pi \mathtt{x}{:}\mathtt{A}_2.\, \mathtt{A} \overset{\leftarrow}{:} \mathtt{type}} \text{ CANON-FAM-PI} \qquad \frac{\Gamma \vdash_\Sigma \mathtt{A} \overset{\rightarrow}{:} \mathtt{type}}{\Gamma \vdash_\Sigma \mathtt{A} \overset{\leftarrow}{:} \mathtt{type}} \text{ CANON-FAM-ATOM}$$

$\boxed{\Gamma \vdash_\Sigma \mathtt{A} \overset{\rightarrow}{:} \mathtt{K}}$

$$\frac{}{\Gamma \vdash_{\Sigma, \mathtt{a} :: \mathtt{K}, \Sigma'} \mathtt{a} \overset{\rightarrow}{:} \mathtt{K}} \text{ ATOM-FAM-CONST}$$

$$\frac{\Gamma \vdash_\Sigma \mathtt{A} \overset{\rightarrow}{:} \Pi \mathtt{x}{:}\mathtt{A}_2.\, \mathtt{K}_1 \qquad \Gamma \vdash_\Sigma \mathtt{M} \overset{\leftarrow}{:} \mathtt{A}_2 \quad \Gamma \vdash_\Sigma [\mathtt{M}/\mathtt{x}]\mathtt{K}_1 \Rightarrow \mathtt{K}\,\mathtt{kind}}{\Gamma \vdash_\Sigma \mathtt{A}\,\mathtt{M} \overset{\rightarrow}{:} \mathtt{K}} \text{ ATOM-FAM-APP}$$

$\boxed{\Gamma \vdash_\Sigma \mathtt{M} \overset{\leftarrow}{:} \mathtt{A}}$

$$\frac{\Gamma \vdash_\Sigma \mathtt{A}_2 \overset{\leftarrow}{:} \mathtt{type} \quad \Gamma, \mathtt{x} : \mathtt{A}_2 \vdash_\Sigma \mathtt{M} \overset{\leftarrow}{:} \mathtt{A}}{\Gamma \vdash_\Sigma \lambda \mathtt{x}{:}\mathtt{A}_2.\, \mathtt{M} \overset{\leftarrow}{:} \Pi \mathtt{x}{:}\mathtt{A}_2.\, \mathtt{A}} \text{ CANON-TERM-LAM}$$

$$\frac{\Gamma \vdash_\Sigma \mathtt{M} \overset{\rightarrow}{:} \mathtt{A} \quad (\mathtt{A} \text{ is } \mathtt{a}) \text{ or } (\mathtt{A} \text{ is } \mathtt{A}'\,\mathtt{M}')}{\Gamma \vdash_\Sigma \mathtt{M} \overset{\leftarrow}{:} \mathtt{A}} \text{ CANON-TERM-ATOM}$$

$\boxed{\Gamma \vdash_\Sigma \mathtt{M} \overset{\rightarrow}{:} \mathtt{A}}$

$$\frac{}{\Gamma, \mathtt{x} : \mathtt{A}, \Gamma' \vdash_\Sigma \mathtt{x} \overset{\rightarrow}{:} \mathtt{A}} \text{ ATOM-TERM-VAR} \qquad \frac{}{\Gamma \vdash_{\Sigma, \mathtt{c} : \mathtt{A}, \Sigma'} \mathtt{c} \overset{\rightarrow}{:} \mathtt{A}} \text{ ATOM-TERM-CONST}$$

$$\frac{\Gamma \vdash_\Sigma \mathtt{M}_1 \overset{\rightarrow}{:} \Pi \mathtt{x}{:}\mathtt{A}_2.\, \mathtt{A}_1 \qquad \Gamma \vdash_\Sigma \mathtt{M}_2 \overset{\leftarrow}{:} \mathtt{A}_2 \quad \Gamma \vdash_\Sigma [\mathtt{M}/\mathtt{x}]\mathtt{A}_1 \Rightarrow \mathtt{A} : \mathtt{type}}{\Gamma \vdash_\Sigma \mathtt{M}_1\,\mathtt{M}_2 \overset{\rightarrow}{:} \mathtt{A}} \text{ ATOM-TERM-APP}$$

A.3 Definition of Well-formedness for Non-Canonical Forms

Typing for non-canonical forms is defined by the following judgements:

- $\Sigma\,\mathtt{sig}$, read "Σ is a well-formed signature"
- $\vdash_\Sigma\ \Gamma\,\mathtt{ctx}$, read "$\Gamma$ is a well-formed context"
- $\Gamma\ \vdash_\Sigma\ \mathtt{K}\,\mathtt{kind}$, read "$\mathtt{K}$ is a well-formed kind"
- $\Gamma\ \vdash_\Sigma\ \mathtt{A} : \mathtt{K}$, read "$\mathtt{A}$ has kind $\mathtt{K}$"
- $\Gamma\ \vdash_\Sigma\ \mathtt{M} : \mathtt{A}$, read "$\mathtt{M}$ has type $\mathtt{A}$"

Because we will only interact with these judgements through the properties below, I'm eliding their definitions.

A.4 Subordination and World Order

DEFINITION A.1: HEAD OF A TYPE FAMILY.

$$\begin{aligned} |\mathtt{a}| &= \mathtt{a} \\ |\mathtt{A}\ \mathtt{M}| &= |\mathtt{A}| \\ |\Pi\,\mathtt{x}{:}\mathtt{A}_2.\,\mathtt{A}| &= |\mathtt{A}| \end{aligned}$$

Observe that this defines a function from type families to family-level constants.

Informally, a type family A is subordinate to a type family B if canonical forms of type A can either appear in canonical forms of type B or appear in canonical indices of the type family B.

DEFINITION A.2: SUBORDINATION.

- Fix a signature Σ. Then a subordination relation between constants, written $\mathtt{a}_1 \preceq \mathtt{a}_2$, is a binary relation between family-level constants in Σ that satisfies the following properties:
  - For all $\mathtt{a}$, $\mathtt{a} \preceq \mathtt{a}$.
  - If $\mathtt{a}_1 \preceq \mathtt{a}_2$ and $\mathtt{a}_2 \preceq \mathtt{a}_3$ then $\mathtt{a}_1 \preceq \mathtt{a}_3$.
  - If $\mathtt{x} : \Pi\,\mathtt{x}_1{:}\mathtt{A}_1 \ldots \Pi\,\mathtt{x}_n{:}\mathtt{A}_n.\,\mathtt{A}$ is in Σ, then $|\mathtt{A}_1| \preceq |\mathtt{A}|$.
  - If $\mathtt{a} : \Pi\,\mathtt{x}_1{:}\mathtt{A}_1 \ldots \Pi\,\mathtt{x}_n{:}\mathtt{A}_n.\,\mathtt{type}$ is in Σ, then $|\mathtt{A}_1| \preceq \mathtt{a}$.
- We then extend subordination to arbitrary type families by taking their head: $\mathtt{A}_1 \preceq \mathtt{A}_2$ iff $|\mathtt{A}_1| \preceq |\mathtt{A}_2|$.

The following definition describes what it means to restrict a context to those entries subordinate to a type family:

DEFINITION A.3: RESTRICTION OF A CONTEXT.

$$\begin{aligned} \cdot|_{\mathtt{A}} &= \cdot \\ (\Gamma, \mathtt{x} : \mathtt{A}_2)|_{\mathtt{A}} &= \Gamma|_{\mathtt{A}}, \mathtt{x} : \mathtt{A}_2 \text{ if } \mathtt{A}_2 \preceq \mathtt{A} \\ &= \Gamma|_{\mathtt{A}} \text{otherwise} \end{aligned}$$

A world $\mathcal{W}$ is a set of contexts.

DEFINITION A.4: WORLD ORDER.

- $\mathcal{W}_1 \leq_{\mathtt{A}} \mathcal{W}_2$ iff for all $\Gamma_1 \in \mathcal{W}_1$, there exists a $\Gamma_2 \in \mathcal{W}_2$ such that $\Gamma_1|_{\mathtt{A}} = \Gamma_2|_{\mathtt{A}}$.
- $\mathcal{W}_1 \equiv_{\mathtt{A}} \mathcal{W}_2$ iff $\mathcal{W}_1 \leq_{\mathtt{A}} \mathcal{W}_2$ and $\mathcal{W}_2 \leq_{\mathtt{A}} \mathcal{W}_1$.

A.5 Properties

I now cite some properties of these judgements.

A.5.1 Assumptions

ASSUMPTION A.5: SUBSTITUTION. *If* $\Sigma\,\mathtt{sig}, \vdash_\Sigma\ \Gamma\,\mathtt{ctx}$, *and* $\Gamma \vdash_\Sigma \mathtt{M_2} : \mathtt{A_2}$, *then*

1. *If* $\Gamma, \mathtt{x}:\mathtt{A_2}, \Gamma' \vdash_\Sigma \mathtt{K}\,\mathtt{kind}$ *then* $\Gamma, [\mathtt{M_2}/\mathtt{x}]\Gamma' \vdash_\Sigma [\mathtt{M_2}/\mathtt{x}]\mathtt{K}\,\mathtt{kind}$.
2. *If* $\Gamma, \mathtt{x}:\mathtt{A_2}, \Gamma' \vdash_\Sigma \mathtt{A}:\mathtt{K}$ *then* $\Gamma, [\mathtt{M_2}/\mathtt{x}]\Gamma' \vdash_\Sigma [\mathtt{M_2}/\mathtt{x}]\mathtt{A} : [\mathtt{M_2}/\mathtt{x}]\mathtt{K}$.
3. *If* $\Gamma, \mathtt{x}:\mathtt{A_2}, \Gamma' \vdash_\Sigma \mathtt{M}:\mathtt{A}$ *then* $\Gamma, [\mathtt{M_2}/\mathtt{x}]\Gamma' \vdash_\Sigma [\mathtt{M_2}/\mathtt{x}]\mathtt{M} : [\mathtt{M_2}/\mathtt{x}]\mathtt{A}$.

ASSUMPTION A.6: CANONIZATION RESULTS ARE CANONICAL. *Assume* $\Sigma\ \overleftarrow{\mathtt{sig}}$ *and* $\vdash_\Sigma\ \Gamma\ \overleftarrow{\mathtt{ctx}}$.

1. *If* $\Gamma \vdash_\Sigma \mathtt{K}\,\mathtt{kind}$ *and* $\Gamma \vdash_\Sigma \mathtt{K} \Rightarrow \mathtt{K'}\,\mathtt{kind}$ *then* $\Gamma \vdash_\Sigma \mathtt{K'}\ \overleftarrow{\mathtt{kind}}$.
2. *If* $\Gamma \vdash_\Sigma \mathtt{K}\ \overleftarrow{\mathtt{kind}}$, $\Gamma \vdash_\Sigma \mathtt{A}:\mathtt{K}$, *and* $\Gamma \vdash_\Sigma \mathtt{A} \Rightarrow \mathtt{A'}:\mathtt{K}$ *then* $\Gamma \vdash_\Sigma \mathtt{A}\ \overleftarrow{:}\ \mathtt{K}$.
3. *If* $\Gamma \vdash_\Sigma \mathtt{A}\ \overleftarrow{:}\ \mathtt{type}$, $\Gamma \vdash_\Sigma \mathtt{M}:\mathtt{A}$, *and* $\Gamma \vdash_\Sigma \mathtt{M} \Rightarrow \mathtt{M'}:\mathtt{A}$ *then* $\Gamma \vdash_\Sigma \mathtt{M}\ \overleftarrow{:}\ \mathtt{A}$.

ASSUMPTION A.7: SOUNDNESS OF CANONICAL FORMS. *If* $\Sigma\,\mathtt{sig}$ *and* $\vdash_\Sigma\ \Gamma\,\mathtt{ctx}$ *then*

1. *If* $\Gamma \vdash_\Sigma \mathtt{K}\ \overleftarrow{\mathtt{kind}}$ *then* $\Gamma \vdash_\Sigma \mathtt{K}\,\mathtt{kind}$
2. *If* $\Gamma \vdash_\Sigma \mathtt{A}\ \overleftarrow{:}\ \mathtt{K}$ *or* $\Gamma \vdash_\Sigma \mathtt{A}\ \overrightarrow{:}\ \mathtt{K}$ *then* $\Gamma \vdash_\Sigma \mathtt{A}:\mathtt{K}$.
3. *If* $\Gamma \vdash_\Sigma \mathtt{M}\ \overleftarrow{:}\ \mathtt{A}$ *or* $\Gamma \vdash_\Sigma \mathtt{M}\ \overrightarrow{:}\ \mathtt{A}$ *then* $\Gamma \vdash_\Sigma \mathtt{M}:\mathtt{A}$.

ASSUMPTION A.8: CANONICAL FORMS EXIST.

1. *If* $\Gamma \vdash_\Sigma \mathtt{K}\,\mathtt{kind}$ *then there exists a* $\mathtt{K'}$ *such that* $\Gamma \vdash_\Sigma \mathtt{K} \Rightarrow \mathtt{K'}\,\mathtt{kind}$.
2. *If* $\Gamma \vdash_\Sigma \mathtt{K}\ \overleftarrow{\mathtt{kind}}$ *and* $\Gamma \vdash_\Sigma \mathtt{A}:\mathtt{K}$ *then there exists a* $\mathtt{A'}$ *such that* $\Gamma \vdash_\Sigma \mathtt{A} \Rightarrow \mathtt{A'}:\mathtt{K}$.
3. *If* $\Gamma \vdash_\Sigma \mathtt{A}\ \overleftarrow{:}\ \mathtt{type}$ *and* $\Gamma \vdash_\Sigma \mathtt{M}:\mathtt{A}$ *then there exists a* $\mathtt{M'}$ *such that* $\Gamma \vdash_\Sigma \mathtt{M} \Rightarrow \mathtt{M'}:\mathtt{A}$.

ASSUMPTION A.9: FACTS ABOUT CANONIZATION.

- *If* $\Gamma \vdash_\Sigma \mathtt{a}\ \overrightarrow{:}\ \Pi\,\mathtt{x_1}{:}\mathtt{A_1}.\ldots\Pi\,\mathtt{x_n}{:}\mathtt{A_n}.\,\mathtt{type}$, *and* $\Gamma \vdash_\Sigma \mathtt{M_i} : [\mathtt{M_{i-1}}/\mathtt{x_{i-1}}]\ldots[\mathtt{M_1}/\mathtt{x_1}]\mathtt{A_i}$ *then* $\Gamma \vdash_\Sigma \mathtt{a}\,\mathtt{M_1}\ \ldots\ \mathtt{M_n} \Rightarrow \mathtt{a}\,\mathtt{M'_1}\ \ldots\ \mathtt{M'_n} : \mathtt{type}$ *where* $\Gamma \vdash_\Sigma \mathtt{M_i} \Rightarrow \mathtt{M'_i} : [\mathtt{M_{i-1}}/\mathtt{x_{i-1}}]\ldots[\mathtt{M_1}/\mathtt{x_1}]\mathtt{A_i}$.
- *If* $\Gamma \vdash_\Sigma \mathtt{A}:\mathtt{type}$ *and* $\Gamma \vdash_\Sigma \mathtt{A} \Rightarrow \mathtt{a}\,\mathtt{M'_1}\ \ldots\ \mathtt{M'_n}:\mathtt{type}$ *then* $\mathtt{A}$ *is* $\mathtt{a}\,\mathtt{M_1}\ \ldots\ \mathtt{M_n}$ *and* $\Gamma \vdash_\Sigma \mathtt{M_i} \Rightarrow \mathtt{M'_i} : [\mathtt{M_{i-1}}/\mathtt{x_{i-1}}]\ldots[\mathtt{M_1}/\mathtt{x_1}]\mathtt{A_i}$ *for all* i.

A.5.2 Lemmas

LEMMA A.10: CANONICAL CLASSIFIERS. *Assume* $\Sigma\ \overleftarrow{\mathtt{sig}}$ *and* $\vdash_\Sigma\ \Gamma\ \overleftarrow{\mathtt{ctx}}$.

1. *If* $\Gamma \vdash_\Sigma \mathtt{M}\ \overleftarrow{:}\ \mathtt{A}$ *or* $\Gamma \vdash_\Sigma \mathtt{M}\ \overrightarrow{:}\ \mathtt{A}$ *then* $\Gamma \vdash_\Sigma \mathtt{A}\ \overleftarrow{:}\ \mathtt{type}$.
2. *If* $\Gamma \vdash_\Sigma \mathtt{A}\ \overleftarrow{:}\ \mathtt{K}$ *or* $\Gamma \vdash_\Sigma \mathtt{A}\ \overrightarrow{:}\ \mathtt{K}$ *then* $\Gamma \vdash_\Sigma \mathtt{K}\ \overleftarrow{\mathtt{kind}}$.

LEMMA A.11: EQUIVALENT WORLDS HAVE THE SAME CANONICAL FORMS. *Assume* $\mathcal{W}_1 \equiv_\mathtt{A} \mathcal{W}_2$. *If* $\Gamma_1 \in \mathcal{W}_1$ *and* $\Gamma_1 \vdash_\Sigma \mathtt{M}\ \overleftarrow{:}\ \mathtt{A}$ *then there exists a* $\Gamma_2 \in \mathcal{W}_2$ *such that* $\Gamma_2 \vdash_\Sigma \mathtt{M}\ \overleftarrow{:}\ \mathtt{A}$; *conversely, if* $\Gamma_2 \in \mathcal{W}_2$ *and* $\Gamma_2 \vdash_\Sigma \mathtt{M}\ \overleftarrow{:}\ \mathtt{A}$ *then there exists a* $\Gamma_1 \in \mathcal{W}_1$ *such that* $\Gamma_1 \vdash_\Sigma \mathtt{M}\ \overleftarrow{:}\ \mathtt{A}$.

LEMMA A.12: INVERSION.

- *If* $\Gamma \vdash_\Sigma \Pi\,\mathtt{x}{:}\mathtt{A_2}.\,\mathtt{A}\ \overleftarrow{:}\ \mathtt{type}$ *then* $\Gamma \vdash_\Sigma \mathtt{A_2}\ \overleftarrow{:}\ \mathtt{type}$ *and* $\Gamma, \mathtt{x}:\mathtt{A_2} \vdash_\Sigma \mathtt{A}\ \overleftarrow{:}\ \mathtt{type}$.

B Adequacy

B.1 Natural Numbers

B.1.1 Syntax

The relevant definitions are

$$\mathsf{N} \quad ::= \quad \mathsf{zero} \mid \mathsf{succ}\ \mathsf{N}$$

```
nat : type.
z : nat.
s : nat -> nat.
```

$$\begin{aligned} \ulcorner \mathsf{zero} \urcorner &= \mathtt{z} \\ \ulcorner \mathsf{succ}\ \mathsf{N} \urcorner &= \mathtt{s}\ \ulcorner \mathsf{N} \urcorner . \end{aligned}$$

Now, the theorem in question is

PROPOSITION B.1: ADEQUACY OF NATURAL NUMBER SYNTAX. *Let Σ be the above signature. Then $\Sigma \overset{\leftarrow}{\mathtt{sig}}$, and there is a bijection between the (informal) natural numbers as defined by the grammar and LF terms $\mathtt{N}$ such that $\cdot \vdash_\Sigma \mathtt{N} \overset{\leftarrow}{:} \mathtt{nat}$.*

Proof. First, we show that the signature is canonical; then, we show that $\ulcorner \mathsf{N} \urcorner$ is a function to canonical LF terms at the appropriate type; finally, we show that for all LF terms such that $\cdot \vdash_\Sigma \mathtt{N} \overset{\leftarrow}{:} \mathtt{nat}$, there exists a unique N such that $\ulcorner \mathsf{N} \urcorner = \mathtt{N}$.

$\Sigma \overset{\leftarrow}{\mathtt{sig}}$ Let $\mathcal{D}_1$ stand for

$$\dfrac{\dfrac{}{\cdot \overset{\leftarrow}{\mathtt{sig}}}\ \text{CANON-SIG-EMPTY} \qquad \dfrac{}{\cdot \vdash_{\cdot} \mathtt{type} \overset{\leftarrow}{\mathtt{kind}}}\ \text{CANON-KIND-TYPE}}{\cdot, \mathtt{nat} : \mathtt{type}\ \overset{\leftarrow}{\mathtt{sig}}}\ \text{CANON-SIG-FAM}$$

Let $\mathcal{D}_2$ stand for

$$\dfrac{\mathcal{D}_1 \quad \dfrac{\dfrac{}{\cdot \vdash_{\cdot, \mathtt{nat}:\mathtt{type}} \mathtt{nat} \overset{\rightarrow}{:} \mathtt{type}}\ \text{ATOM-FAM-CONST}}{\cdot \vdash_{\cdot, \mathtt{nat}:\mathtt{type}} \mathtt{nat} \overset{\leftarrow}{:} \mathtt{type}}\ \text{CANON-FAM-ATOM}}{\cdot, \mathtt{nat} : \mathtt{type}, \mathtt{z} : \mathtt{nat}\ \overset{\leftarrow}{\mathtt{sig}}}\ \text{CANON-SIG-TERM}$$

Then the derivation is

$$\dfrac{\dfrac{\dfrac{}{\cdot \vdash_{\Sigma_1, \mathtt{nat}:\mathtt{type}, \Sigma_2} \mathtt{nat} \overset{\rightarrow}{:} \mathtt{type}}\ \text{ATOM-FAM-CONST}}{\cdot \vdash_{\Sigma_1, \mathtt{nat}:\mathtt{type}, \Sigma_2} \mathtt{nat} \overset{\leftarrow}{:} \mathtt{type}}\ \text{CANON-FAM-ATOM} \qquad \dfrac{\dfrac{}{\_ : \mathtt{nat} \vdash_{\Sigma_1, \mathtt{nat}:\mathtt{type}, \Sigma_2} \mathtt{nat} \overset{\rightarrow}{:} \mathtt{type}}\ \text{ATOM-FAM-CONST}}{\_ : \mathtt{nat} \vdash_{\Sigma_1, \mathtt{nat}:\mathtt{type}, \Sigma_2} \mathtt{nat} \overset{\leftarrow}{:} \mathtt{type}}\ \text{CANON-FAM-ATOM}}{\cdot \vdash_{\Sigma_1, \mathtt{nat}:\mathtt{type}, \Sigma_2} \Pi \_ : \mathtt{nat}. \mathtt{nat} \overset{\leftarrow}{:} \mathtt{type}}\ \text{CANON-FAM-PI}}{\cdot, \mathtt{nat} : \mathtt{type}, \mathtt{z} : \mathtt{nat}, \mathtt{s} : \Pi \_ : \mathtt{nat}. \mathtt{nat}\ \overset{\leftarrow}{\mathtt{sig}}}\ \text{CANON-SIG-TERM}$$

where $\Sigma_1 = \cdot$ and $\Sigma_2 = \mathtt{nat} : \mathtt{type}, \mathtt{z} : \mathtt{nat}$. In the future, we will leave the pattern-matching involved in dividing the signature to the reader.

For all N, there exists a unique $\mathtt{N}$ such that $\ulcorner \mathsf{N} \urcorner = \mathtt{N}$ and $\cdot \vdash_\Sigma \mathtt{N} \overset{\leftarrow}{:} \mathtt{nat}$. The proof is by structural induction on N.

- Case for zero. Take $\mathtt{N}$ to be $\mathtt{z}$; then $\ulcorner \mathsf{zero} \urcorner = \mathtt{z}$ by definition, establishing existence exists. To show uniqueness, assume some other $\mathtt{Y}$ such that $\ulcorner \mathsf{zero} \urcorner = \mathtt{Y}$. By inversion, the only case of $\ulcorner \cdot \urcorner$ that applies is the one for zero, and in this case $\mathtt{y} = \mathtt{z}$. Then the following derivation proves that $\mathtt{z}$ is canonical at the appropriate type:

$$\dfrac{\dfrac{}{\cdot \vdash_{\Sigma_1, \mathtt{z}:\mathtt{nat}, \Sigma_2} \mathtt{z} \overset{\rightarrow}{:} \mathtt{nat}}\ \text{ATOM-TM-CONST}}{\cdot \vdash_\Sigma \mathtt{z} \overset{\leftarrow}{:} \mathtt{nat}}\ \text{CANON-TM-ATOM}$$

- Case for $\mathsf{succ}\ \mathsf{N}'$. To show: for all N', if there exists a unique $\mathtt{N}'$ such that $\ulcorner \mathsf{N}' \urcorner = \mathtt{N}'$ and $\mathcal{D}$ derives $\cdot \vdash_\Sigma \mathtt{N}' \overset{\leftarrow}{:} \mathtt{nat}$, then there exists a unique $\mathtt{N}$ such that $\ulcorner \mathsf{succ}\ \mathsf{N}' \urcorner = \mathtt{N}$ and $\cdot \vdash_\Sigma \ulcorner \mathsf{succ}\ \mathsf{N} \urcorner \overset{\leftarrow}{:} \mathtt{nat}$.

  Make the assumptions. Then take $\mathtt{N} = \mathtt{s}\ \mathtt{N}'$; by definition, $\ulcorner \mathsf{succ}\ \mathsf{N}' \urcorner = \mathtt{s}\ \ulcorner \mathsf{N}' \urcorner = \mathtt{s}\ \mathtt{N}'$, so such an $\mathtt{N}$ exists. Now take some other $\mathtt{N}''$ such that $\ulcorner \mathsf{succ}\ \mathsf{N} \urcorner = \mathtt{N}''$; then, there is only one case of the encoding that applies to $\mathsf{succ}\ \mathsf{N}'$, so $\mathtt{N}'' = \mathtt{succ}\ \ulcorner \mathsf{N}' \urcorner$. By assumption $\ulcorner \mathsf{N}' \urcorner = \mathtt{N}'$ uniquely, so we get the same encoding in both cases. This shows uniqueness.

  Now we must show $\cdot \vdash_\Sigma \ulcorner \mathsf{succ}\ \mathsf{N}' \urcorner \overset{\leftarrow}{:} \mathtt{nat}$, or, equivalently, $\cdot \vdash_\Sigma \mathtt{s}\ \mathtt{N}' \overset{\leftarrow}{:} \mathtt{nat}$. Here is a derivation:

$$\dfrac{\dfrac{}{\cdot \vdash_\Sigma \mathtt{s} \overset{\rightarrow}{:} \Pi \_ {:}: \mathtt{nat}.\, \mathtt{nat}}\ \text{ATOM-TERM-CONST} \qquad \begin{array}{c} \mathcal{D} \\ \cdot \vdash_\Sigma \mathtt{N}' \overset{\leftarrow}{:} \mathtt{nat} \end{array} \qquad \begin{array}{c} \vdots \\ \cdot \vdash_\Sigma [\mathtt{N}'/\_]\mathtt{nat} \Rightarrow \mathtt{nat} : \mathtt{type} \end{array}}{\dfrac{\cdot \vdash_\Sigma \mathtt{s}\ \mathtt{N}' \overset{\rightarrow}{:} \mathtt{nat}}{\cdot \vdash_\Sigma \mathtt{s}\ \mathtt{N}' \overset{\leftarrow}{:} \mathtt{nat}}\ \text{CANON-TERM-ATOM}}\ \text{ATOM-TERM-APP}$$

  The canonization premise follows from ASSUMPTION A.9.

For all $\mathtt{N}$ such that $\cdot \vdash_\Sigma \mathtt{N} \overset{\leftarrow}{:} \mathtt{nat}$, there exists a unique N such that $\ulcorner \mathsf{N} \urcorner = \mathtt{N}$. We first invert $\cdot \vdash_\Sigma \mathtt{N} \overset{\leftarrow}{:} \mathtt{nat}$ to discover the possible canonical forms. This derivation must have been derived using CANON-TERM-ATOM, as $\mathtt{nat}$ is a constant, so we must have derived $\cdot \vdash_\Sigma \mathtt{N} \overset{\rightarrow}{:} \mathtt{nat}$. How could we have derived this?

- Because the context is empty, we cannot have used ATOM-TERM-VAR.
- We might have used ATOM-TERM-CONST, but in this case, based on the signature, $\mathtt{N}$ must be $\mathtt{z}$.
- We might have used ATOM-TERM-APP, in which case we derived $\cdot \vdash_\Sigma \mathtt{N}_1 \overset{\rightarrow}{:} \Pi\, \mathtt{x}{:}\mathtt{A}_2.\, \mathtt{A}_1$, $\cdot \vdash_\Sigma \mathtt{N}_2 \overset{\leftarrow}{:} \mathtt{A}_2$, and $\cdot \vdash_\Sigma [\mathtt{N}_2/\mathtt{x}]\mathtt{A}_1 \Rightarrow \mathtt{nat} : \mathtt{type}$. Then, by ASSUMPTION A.9, $\mathtt{A}_1$ must be $\mathtt{nat}$ (note that we use LEMMA A.10, LEMMA A.12, ASSUMPTION A.7, and ASSUMPTION A.5 to satisfy the premise). Thus, we must consider how we could have derived $\cdot \vdash_\Sigma \mathtt{N}_1 \overset{\rightarrow}{:} \Pi\, \mathtt{x}{:}\mathtt{A}_2.\, \mathtt{nat}$:
  - Again, we cannot have used ATOM-TERM-VAR.
  - We might have used ATOM-TERM-CONST, but based on the signature $\mathtt{N}_1$ is $\mathtt{s}$ and $\mathtt{A}_2$ is $\mathtt{nat}$.
  - We cannot have used ATOM-TERM-APP, as this would require a premise of $\cdot \vdash_\Sigma \mathtt{N}_1' \overset{\rightarrow}{:} \Pi\, \mathtt{y}{:}\mathtt{B}_1.\, \Pi\, \mathtt{x}{:}\mathtt{A}_2'.\, \mathtt{nat}$. However, there are no $\mathtt{N}_1'$ such that $\cdot \vdash_\Sigma \mathtt{N}_1' \overset{\rightarrow}{:} \Pi\, \mathtt{y}_1{:}\mathtt{B}_1 \ldots \Pi\, \mathtt{y}_k{:}\mathtt{B}_k.\, \Pi\, \mathtt{x}{:}\mathtt{A}_2'.\, \mathtt{nat}$ for $k \geq 0$. To prove this, we assume one exists and derive a contradiction. One rule must have been used to derive $\cdot \vdash_\Sigma \mathtt{N}_1' \overset{\rightarrow}{:} \Pi\, \mathtt{y}_1{:}\mathtt{B}_1 \ldots \Pi\, \mathtt{y}_k{:}\mathtt{B}_k.\, \Pi\, \mathtt{x}{:}\mathtt{A}_2'.\, \mathtt{nat}$. It cannot have been ATOM-TERM-VAR because the context is empty; it cannot have been ATOM-TERM-CONST because there are no

constants of this form in the signature. The only rule that can have applied is ATOM-TERM-APP. However, this rule has as its premise a derivation $\cdot \vdash_\Sigma \mathtt{N}_1'' \stackrel{\rightarrow}{:} \Pi\, \mathtt{y}_0{:}\mathtt{B}_0.\, \Pi\, \mathtt{y}_1{:}\mathtt{B}_1'. \ldots \Pi\, \mathtt{y}_k{:}\mathtt{B}_k'.\, \Pi\, \mathtt{x}{:}\mathtt{A}_2''.\, \mathtt{nat}$, so we get a contradiction by induction. Intuitively, there are no variables or constants at which we can root the applications.

Thus, either $\mathtt{N}$ is $\mathtt{z}$, or $\mathtt{N}$ is $\mathtt{s}\ \mathtt{N}_2$, in which case we derived as a strict subderivation that $\cdot \vdash_\Sigma \mathtt{N}_2 \stackrel{\leftarrow}{:} \mathtt{nat}$. We now prove the theorem for each of these cases.

- $\mathtt{N}$ is $\mathtt{z}$. $\ulcorner \mathsf{zero} \urcorner = \mathtt{z}$, so there exists an N such that $\ulcorner \mathsf{N} \urcorner = \mathtt{z}$. To show uniqueness, assume some other N' such that $\ulcorner \mathsf{N}' \urcorner = \mathtt{z}$. By inversion on the definition of the encoding, N' is zero.

- $\mathtt{N}$ is $\mathtt{s}\ \mathtt{N}_2$, in which case we derived as a strict subderivation that $\cdot \vdash_\Sigma \mathtt{N}_2 \stackrel{\leftarrow}{:} \mathtt{nat}$. By induction on the subderivation, there exists a unique number N_2 such that $\ulcorner \mathsf{N}_2 \urcorner = \mathtt{N}_2$.

  Now take N to be $\mathsf{succ}\ \mathsf{N}_2$, which is syntactically correct because N_2 is. $\ulcorner \mathsf{N} \urcorner = \ulcorner \mathsf{succ}\ \mathsf{N}_2 \urcorner = \mathtt{s}\ \ulcorner \mathsf{N}_2 \urcorner = \mathtt{s}\ \mathtt{N}_2$ by definition of the encoding, so a preimage exists. To show uniqueness, assume another N' such that $\ulcorner \mathsf{N}' \urcorner = \mathtt{s}\ \mathtt{N}_2$. By inversion on the definition of the encoding, N' is $\mathsf{succ}\ \mathsf{N}_2'$, where $\ulcorner \mathsf{N}_2' \urcorner = \mathtt{N}_2$. But N_2 is unique number such that $\ulcorner \mathsf{N}_2 \urcorner$ is $\mathtt{N}_2$, so $\mathsf{N}_2' = \mathsf{N}_2$. Thus, N' is $\mathsf{succ}\ \mathsf{N}_2$, establishing uniqueness.

□

B.1.2 `sum` Judgement

B.2 System F

B.2.1 Syntax

B.2.2 Static Semantics

B.2.3 Dynamic Semantics