Week 4 Lecture Notes

Jason Koenig

10/7/2013 and 10/9/2013

1 Dependent Types

1.1 Structural Foundation

Dependent types are *families* of types. Recall from last week the structural setup, which differs from IPL in that we need to express *dependency* between terms and types in the context and the preceeding terms. Atomic judgements are of the form

contexts / closed types:	$\Gamma\ \mathsf{ctx}$
	$\Gamma \equiv \Gamma'$
open types / families of types:	$\Gamma \vdash A\ \mathsf{type}$
	$\Gamma \vdash A \equiv A'$
elements of types:	$\Gamma \vdash M : A$
	$\Gamma \vdash M \equiv M' : A$

The symbol $\equiv$ denotes what we will interpret as *definitional equality*. We denote the *empty context* by $\cdot$ when we need to. The introduction rules for contexts are:

$$\frac{}{\cdot\ \mathsf{ctx}} \qquad \frac{\Gamma\ \mathsf{ctx} \quad \Gamma \vdash A\ \mathsf{type}}{\Gamma, x : A\ \mathsf{ctx}}$$

Thus we have some notion of *dependence*; it allows us to make sense of expressions like $x : \mathsf{Nat}, y : \mathsf{Seq}(x) \vdash \cdots$, which was impossible before.

$$\frac{}{\cdot \equiv \cdot\ \mathsf{ctx}} \qquad \frac{\Gamma \equiv \Gamma' \quad \Gamma \vdash A \equiv A'}{\Gamma, x : A \equiv \Gamma', x : A'}$$

The following rule corresponds with reflexivity:

$$\frac{}{\Gamma, x : A, \Delta \vdash x : A}$$

The following rules (one for each judgement J) correspond with weakening:

$$\frac{\Gamma, \Delta \vdash J \quad \Gamma \vdash A\ \mathsf{type}}{\Gamma, x : A, \Delta \vdash J}$$

Exercise. What are the corresponding rules for exchange and contraction?

The following rule, called *substitution* or *instantiation*, corresponds with transitivity:

$$\frac{\Gamma, x : A, \Delta \vdash J \quad \Gamma \vdash M : A}{\Gamma[M/x]\Delta \vdash [M/x]J}$$

The following rules together are called *functionality*

$$\frac{\Gamma, x : A, \Delta \vdash N : B \quad \Gamma \vdash M \equiv M' : A}{\Gamma[M/x]\Delta \vdash [M/x]N \equiv [M'/x]N : [M/x]B}$$

$$\frac{\Gamma, x : A, \Delta \vdash B\ \mathsf{type} \quad \Gamma \vdash M \equiv M' : A}{\Gamma[M/x]\Delta \vdash [M/x]B \equiv [M'/x]B}$$

Finally, the following rules are *type equality*, which tell us that definitionally equal types classify the same things:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash A \equiv A'}{\Gamma \vdash M : A'} \qquad \frac{\Gamma \vdash M \equiv M' : A \quad \Gamma \vdash A \equiv A'}{\Gamma \vdash M \equiv M' : A'}$$

1.2 Negative Dependent Types

In dependent type theory, the negative types need to be generalized to express dependency. The type $\top$ does not change. There are new forms for $\wedge$ and $\vee$, however:

IPL	Type		Dependent Type	Quantifier
$A \wedge B$	$A \times B$	$\rightarrow$	$\Sigma x : A.B$	$\exists x : A.B$
$A \supset B$	B^A or $A \rightarrow B$	$\rightarrow$	$\Pi x : A.B$	$\forall x : A.B$

Before we present the rules governing these types, it is good to get some intuition for what they represent. The dependent type $\Sigma x : A.B$, called the *sum*, *dependent product*, or *sigma-type*. Here unfortunately terminology becomes muddled, as Σ properly generalizes the product. The Σ type corresponds to (constructive) existential quantification, $\exists x : A.B$.

The type $\Pi x : A.B$ is called the *dependent product* (ambiguously with with above), or *pi-type*. This generalizes functions from $A \rightarrow B$ in that the type of the result of an application can depend on exactly which value the function is applied to. This corresponds to universal quantification $\forall x : A.B$.

Example: We can form complex types from these connectives and the families we have seen. For example, we can form the type $\Pi x : \mathsf{Nat}.\Sigma y : \mathsf{Nat}.\mathsf{Id}_{\mathsf{Nat}}(y, \mathrm{succ}(x))$. By the types-propositions correspondence, this is the proposition $\forall x : \mathsf{Nat}.\exists y : \mathsf{Nat}.y =_{\mathsf{Nat}} \mathrm{succ}(x)$, i.e. for every natural number, there is a successor. We can think of the as inhabited by proofs that the proposition is true. We can also think of elements of this type as terms which take a natural number x, and produce a term which is a proof that that particular natural has a successor.

Example: A less logical example is the type $\Pi x : \mathsf{Nat}.\mathrm{Seq}(x)$. This represents the type of functions which result in a length n sequence when applied to the value n, as one might do if "allocating" a sequence. This might be useful when programming, as the type encodes more information about a value than a non-dependent type could.

One important aspect of the quantifier view of these types is that they generalize classical logic in that they merge domains of quantification and proposition. In classical logic, and especially first order logic, these are completely distinct. With the identification of propositions and types, we can quantify not just over data (such as Nat), but also over proofs of propositions by letting A be a "proposition"-ish type like $\mathsf{Id}_{\mathsf{Nat}}(x, y)$.

This identification of dependent types with quantifiers only holds if you take the quantifiers as *constructive*. An element of type $\Sigma x : A.B$ consists of a *pair* $\langle x, y \rangle$ such that x is an witness of the existential, and y is a proof that that element means the condition. A constructive existential requires there actually to be an explicit element, which the proof must give on its own. This means one cannot use certain kinds of proofs from classical logic. Some of these proofs roughly show: $(\forall x.P) \supset \bot$, so they conclude that $\exists x.P$, but nowhere in the proof can we actually find such an x. Similarly, but less counterintuitively, a proof of an universal quantifier is a map from elements to proofs for those specific elements, i.e. a function. We cannot prove $\forall x.P$ by showing $(\exists x.P) \supset \bot$.

The idea with constructivity is then that if you have proved $(\forall x.P) \supset \bot$, then just let that be the proof. Constructivity amounts to a certain carefulness, where we avoid the classical convention that $\neg\exists \iff \forall$ and $\neg\forall \iff \exists$. By making this distinction, we get not only the ability to regard proofs as programs, but also the rich structure of HoTT, which would otherwise collapse.

1.2.1 Pi-types

The formation and introduction rules for pi-types are:

$$\frac{\Gamma \vdash A\ \mathsf{type} \quad \Gamma, x : A \vdash B\ \mathsf{type}}{\Gamma \vdash \Pi x : A.B\ \mathsf{type}}\ \Pi\text{-F} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : M : \Pi x : A.B}\ \Pi\text{-I}$$

Note that the introduction form is almost the same except that x is bound when forming the type B. The introduction rule is exactly the same as in IPL, except that the type B depends on x like M does. This dependency of B motivates the elimination rule:

$$\frac{\Gamma \vdash M : \Pi x : A.B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : [N/x]B}\ \Pi\text{-E}$$

which is the same as IPL except for the substitution of N, the actual argument, into the result type. Thus the result type can vary with the actual argument, which cannot happen in regular type theory.

We also have the (β) and (η) rules:

$$\begin{aligned} (\lambda x.M)N &\equiv [N/x]M \quad (\beta) \\ (\lambda x.Mx) &\equiv M \quad (\eta) \quad (\text{when } x \text{ not free in } M) \end{aligned}$$

1.2.2 Sigma-types

The formation and introduction rules for sigma-types are:

$$\frac{\Gamma \vdash A\ \mathsf{type} \quad \Gamma, x : A \vdash B\ \mathsf{type}}{\Gamma \vdash \Sigma x : A.B\ \mathsf{type}}\ \Sigma\text{-F} \qquad \frac{\Gamma \vdash M : A \quad \Gamma, x : A \vdash N : [M/x]B}{\Gamma \vdash \langle M, N \rangle : \Sigma x : A.B}\ \Sigma\text{-I}$$

The addition here is that the second component can depend on the first. The elimination rules are what one expects, with the addition of the dependence of the type of second component.

$$\frac{\Gamma \vdash M : \Sigma x : A.B}{\Gamma \vdash \mathsf{fst}(M) : A}\ \Sigma\text{-E}_1 \qquad \frac{\Gamma \vdash M : \Sigma x : A.B}{\Gamma \vdash \mathsf{snd}(M) : [\mathsf{fst}(M)/x]B}\ \Sigma\text{-E}_2$$

The (β) and (η) rules are the familiar:

$$\begin{aligned} \mathsf{fst}(\langle M, N \rangle) &\equiv M \quad (\beta_1) \\ \mathsf{snd}(\langle M, N \rangle) &\equiv N \quad (\beta_2) \\ \langle \mathsf{fst}(M), \mathsf{snd}(M) \rangle &\equiv M \quad (\eta) \end{aligned}$$

1.3 Positive Dependent Types

With the positive types, the dependency does not affect the form of the types, but rather their elimination forms. The issue arises because we need a "join point" in the elimination. For example, C is the join point in the elimination of $\vee$ in IPL because it needs to hold in both cases:

$$\frac{\Gamma, x : A \vdash N : C \quad \Gamma, y : B \vdash P : C \quad \Gamma \vdash M : A + B}{\Gamma \vdash \mathsf{case}(M, x.N, y.P) : C} \vee\text{-E}_{\text{IPL}}$$

In IPL, there is no issue with not making C dependent because types are fixed: there is nothing that can vary about C in either branch. In dependent type theory, this is no longer the case.

1.3.1 Sum Types

To motivate why we need dependency in C, we consider the sum (i.e. disjunction). First we introduce the booleans, a simple case of the $(+)$ type. Let us define the boolean type 2 as $1 + 1$, where 1 is the unit type, and the values $\mathsf{tt} = \mathsf{inl}(\langle\rangle) : 2$ and $\mathsf{ff} = \mathsf{inr}(\langle\rangle) : 2$.

Consider the rather trivial proposition (written as a type) $\Pi x : 2.(\mathsf{Id}_2(x, \mathsf{ff})) + (\mathsf{Id}_2(x, \mathsf{tt}))$. In order to prove this, we need to perform a case analysis on x. In particular, the proposition we actually prove is different in each case:
1. False case: prove $(\mathsf{Id}_2(\mathsf{ff}, \mathsf{ff})) + (\mathsf{Id}_2(\mathsf{ff}, \mathsf{tt}))$ by left injection of reflexivity
2. True case: prove $(\mathsf{Id}_2(\mathsf{tt}, \mathsf{ff})) + (\mathsf{Id}_2(\mathsf{tt}, \mathsf{tt}))$ by right injection of reflexivity.
We see than if we case analyze on x, then the actual proposition we need to prove in each case is the goal we are trying to prove, specialized to the particular case we are in. The following rule codifies this:

$$\frac{\Gamma, x : A \vdash N : [\mathsf{inl}(x)/z]C \quad \Gamma, y : B \vdash P : [\mathsf{inr}(y)/z]C \quad \Gamma \vdash M : A + B \quad \Gamma, z : A + B \vdash C\ \mathsf{type}}{\Gamma \vdash \mathsf{case}[z.C](M, x.N, y.P) : [M/z]C} \vee\text{-E}$$

In the case(...) construct, the part $[z.C]$ is called the *motive*, because it is the motivation for performing the case analysis in the first place. What is different here from IPL is that C is parametric in $z : A + B$. The reason that this is necessary is that we need to be able to substitute the actual case we are analyzing (either inl or inr) in each branch into the type, so that our proof can vary. The case analysis proves C specialized to M, which is the term M flowing into the *type* C. The parametricity in z allows this dependency.

We can specialize this rule to the booleans, to obtain a new construct if:

$$\frac{\Gamma \vdash N : [\mathsf{tt}\,/z]C \quad \Gamma \vdash P : [\mathsf{ff}\,/z]C \quad \Gamma \vdash M : 2 \quad \Gamma, z : 2 \vdash C\ \mathsf{type}}{\Gamma \vdash \mathsf{if}[z.C](M, N, P) : [M/z]C}\ \vee\text{-E}$$

where the terms N and P do not bind any variables because by (η), the variables in the $\vee$-E rule are both just the null tuple, and thus don't actually need to be bound.

This presentation seems justified in the logical interpretation, but in programming it can lead to some unexpected results. In most programming languages, $\mathsf{if}(x, 17, tt)$ would not be well typed, as the true branch has type Nat and the false branch has type 2. With the tools of dependent type theory, however, we can give this term a type. If we posit the existence of conditionals in the type level, which we have not formalized yet, we can imagine the typing:

$$\mathsf{if}(M, 17, \mathsf{tt}) : \mathsf{if}(M, \mathsf{Nat}, 2)$$

for some suitable if construct at the type level, and noting that the "2" is a type name, not $s(s(z))$. Even though the two branches have different simple types, $\mathsf{if}(M, \mathsf{Nat}, 2)$ represents a join point for the two expressions. In normal programming languages, the join point type is not allowed to depend on any term, let alone the actual branch that was taken. This example then seems ill-typed, but in fact $17 : \mathsf{if}(\mathsf{tt}, \mathsf{Nat}, 2)$ and $\mathsf{tt} : \mathsf{if}(\mathsf{ff}, \mathsf{Nat}, 2)$, which are the critical premises in the $\vee$-E rule. This kind of construct is important when encoding $A + B$ using sigma types.

Finally, we can take some β and η rules as well, which mirror the non-dependent case. Here we omit the motive as it plays no role in the rules:

$$\begin{aligned}
&\mathsf{case}(\mathsf{inl}(M), x.N, y.P) \equiv [M/x]N && \beta_1 \\
&\mathsf{case}(\mathsf{inr}(M), x.N, y.P) \equiv [M/y]P && \beta_2 \\
&\mathsf{case}(M, x.[\mathsf{inl}(x)/z]P, y.[\mathsf{inr}(y)/z]P) \equiv [M/z]P && \eta
\end{aligned}$$

1.3.2 Natural Numbers

Like sum types, there is a relatively straightforward generalization of the elimination rule for naturals in terms of a recursor augmented with a motive:

$$\frac{\Gamma \vdash M : \mathsf{Nat} \quad \Gamma, z : \mathsf{Nat} \vdash C\ \mathsf{type} \quad \Gamma \vdash N : [0/z]C \quad \Gamma, x : \mathsf{Nat}, y : [x/z]C \vdash P : [s(x)/z]C}{\Gamma \vdash \mathsf{rec}[z.C](M, N, x.y.P) : [M/z]C}\ \mathsf{Nat\text{-}E}$$

Just as in the sum case, we can interpret the recursor as a proof of C for the specific natural M, given N which is a proof for zero and P which proves C for the successor

of some natural. In Gödel's T, we could show that omitting the variable $x : Nat$ was possible because in the presence of products we could add it if necessary. In this formulation, however, the type of y is a proof of C for some natural, so we need to give that natural a name to form the type.

We also have the (β) and (η) rules as expected:

$$\mathsf{rec}[z.C](0, N, x.y.P) \equiv N \quad \beta_1$$

$$\mathsf{rec}[z.C](s(M), N, x.y.P) \equiv [M, \mathsf{rec}[z.C](M, N, x.y.P)/x, y]P \quad \beta_2$$

$$\frac{[0/w]M \equiv N \quad [s(w)/w]M \equiv [w, M/x, y]P}{M \equiv \mathsf{rec}[z.C](w, N, x.y.P)} \quad \eta$$

1.3.3 Sigma elimination

We can characterize the sigma type elimination without fst and snd using a "degenerate form" of pattern matching where we have only one case:

$$\frac{\Gamma \vdash M : \Sigma x : A.B \quad \Gamma, z : \Sigma x : A.B \vdash C\ \mathsf{type} \quad \Gamma, x : A, y : B \vdash P : [\langle x, y\rangle / z]C}{\Gamma \vdash \mathsf{split}[z.C](M, x.y.P) : [M/z]C}\ \Sigma\text{-E}$$

We can take the following beta rule:

$$\mathsf{split}[z.C](\langle M_1, M_2\rangle, x.y.P) \equiv [M_1, M_2/x, y]P \quad (\beta)$$

We leave as an exercise showing that one can implement split from fst and snd. We can also show the converse, that fst and snd are definable in terms of split.

2 Identity Types

With this background, we can begin to talk meaningfully about the identity type. We say that the type $\mathsf{Id}_A(M, N)$ is the identity type of M and N in A, where $A\,\mathsf{type}$, $M : A$, and $N : A$. We can think of terms of this type as *proofs* that M and N are equal as elements of A. Crucially, we need dependence to even form this type. The formation rule is:

$$\frac{\Gamma \vdash A\,\mathsf{type} \quad \Gamma \vdash M : A \quad \Gamma \vdash N : A}{\Gamma \vdash \mathsf{Id}_A(M, N)\,\mathsf{type}}\ \mathsf{Id}\text{-F}$$

This type may also be written $M =_A N$. Confusingly, it may also be referred to as "propositional equality," to distinguish it from definitional equality. The simplest

element of this type is a proof of reflexivity, which has the introduction rule:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{refl}_A(M) : \mathsf{Id}_A(M, M)}\ \mathsf{Id\text{-}I}$$

The elimination rule is:

$$\frac{\Gamma \vdash P : \mathsf{Id}_A(M, N) \quad \Gamma, x{:}A, y{:}A, z{:}\mathsf{Id}_A(x, y) \vdash C\ \mathsf{type} \quad \Gamma, x{:}A \vdash Q : [x, x, \mathsf{refl}_A(x)/x, y, z]C}{\Gamma \vdash \mathsf{J}[x.y.z.C](P, x.Q) : [M, N, P/x, y, z]C}\ \mathsf{Id\text{-}E}$$

One way to think of the identity type for a particular A is as an inductively generated family of types, where the induction is taken simultaneously over the two terms and the proof that they are equal (corresponding to x, y, and z in the rule). This view leads us to call the elimination rule *path induction*, where elements of type Id are "paths." Then M and N are the *endpoints* of this path. We call this an induction because we can prove C for any two endpoints and proof of their equality if we can prove C generically for some endpoint using Q. We need to consider a generic x in Q because the whole family Id_A is inductively generated at once, so we can't fix the endpoints M and N while proving the "inductive step." Further, because we only have one introduction rule, we can reason based on the fact that every path is just reflexivity, so Q can make this assumption.

J also admits a β-like rule:

$$\mathsf{J}[x.y.z.C](\mathsf{refl}_A(M); x.Q) \equiv [M/x]Q : [M, M, \mathsf{refl}_A(M)/x, y, z]C \quad (\beta)$$

2.1 Identity Types as Equivalence Relations

If Id is supposed to be equality, then we would like it to satisfy the three conditions of an equivalence relation: reflexivity, symmetry, and transitivity. Reflexivity is already part of the definition, but the other two properties are not part of the definition. We would like *proofs* that identity types satisfy these properties. Because we are working in a proof-relevant system, these proofs can also be viewed as enriched functional programs. Symmetry states that $x = y \implies y = x$, so a proof of symmetry is a simply map from $\mathsf{Id}_A(x, y)$ to $\mathsf{Id}_A(y, x)$. We can write the type of such a map formally as $\Pi x, y{:}A.\, \mathsf{Id}_A(x, y) \to \mathsf{Id}_A(y, x)$ (which is shorthand for $\Pi x, y{:}A.\, \Pi p{:}\mathsf{Id}_A(x, y).\, \mathsf{Id}_A(y, x)$). A proof of symmetry is just a *program* which has this type:

$$\mathrm{sym}_A := \lambda x{:}A.\lambda y{:}A.\lambda p{:}\mathsf{Id}_A(x, y).\mathsf{J}[x.y.z.\mathsf{Id}_A(y, x)](p; x.\mathsf{refl}_A(x))$$

The construction of symmetry is a function (using three λ's) whose body just immediately invokes path induction. The motive just states the overall conclusion, and

the proof Q is just the only way we have to generate elements of Id_A. The reason this proof is simple is any path is just reflexivity. The relation defined by the identity type is effectively the diagonal relation which only relates things to themselves, about which it is easy to prove symmetry.

Transitivity is more involved. Here we have not one, but two equality proofs, but J will only let us inspect one at a time. Like symmetry, we write trans_A as a map from elements and proofs of their equality to another proof of equality:

$$\begin{aligned}\mathrm{trans}_A :=& \lambda x, y, z{:}A.\ \lambda u : \mathsf{Id}_A(x, y).\ \lambda v{:}\mathsf{Id}_A(y, z).\\ &(\mathsf{J}[m.n.u.\ \mathsf{Id}_A(n, z) \to \mathsf{Id}_A(m, z)](u; m.\lambda v.v))(v)\end{aligned}$$

The way to understand this is that we need to inductively define a function, which will show that if y is equal to z, then x is equal to z. Then we can apply this function to the proof v, and obtain the desired result. For Q, we see that the motive degenerates to $\mathsf{Id}_A(m, z) \to \mathsf{Id}_A(m, z)$, so we can just write the identity function.

These specific proofs of symmetry and transitivity induce a kind of β-like behavior for definitional equality. In particular, due to the β rule for J, we see that

$$\mathrm{sym}_A(M)(M)(\mathsf{refl}_A(M)) \equiv \mathsf{refl}_A(M)$$

Further,

$$\mathrm{trans}_A(X)(X)(Z)(\mathsf{refl}_A(X))(Q) \equiv Q$$

because the induction J is defined on u, and thus when u is a reflexivity the whole J term reduces to the identity function. This definitional equivalence depends on the *how* we proved trans_A. For example, a different proof might not make the same equivalences, which can introduce dependencies between a proof and the exact definition of its lemmas. From a programming perspective, this is anti-modular.

2.2 Simple Functionality

In addition being an equivalence relation, we might hope that maps respect equality. Suppose we have a map F from A to B (so $x{:}A \vdash F : B$). In the non-dependent case, B will not depend on x, so we can can ask for some term to satisfy:

$$x, y{:}A, u{:}\mathsf{Id}_A(x, y) \vdash \_ : \mathsf{Id}_B(Fx, Fy)$$

We introduce the term $\mathsf{ap}Fu$ to be the lifting of F from a map between terms to a map between paths. ap may also be known as the "functorial action." We can define ap using a path induction as:

$$\mathsf{ap}\ F\ u := \mathsf{J}[x.y.\_.\mathsf{Id}_B(Fx, Fy)](u; x.\mathsf{refl}_B(Fx))$$

2.3 Transportation

If we have a dependent map, then the above is no longer sufficient. In this case, we have that $z{:}A \vdash B\ \mathsf{type}$ and $x{:}A \vdash F : B$ as before. Crucially different is that B may depend on x, so Fx and Fy won't even necessarily have the same type! Thus we can't even form $\mathsf{Id}_B(Fx, Fy)$. If $x :\equiv y$, then we would know that the two destination types are the same by substitution, but we only have that x is equal to y *propositionally*. We would expect that $B[x]$ and $B[y]$ to be *related* in some way, as but we do not go as far as to say the two spaces are the same.

What we will do is say that if two elements x and y have a path p in A between x and y (i.e. $p : \mathsf{Id}_A(x, y)$), then there is a lifting of this path, p_*, to act as a transport map between $[x/z]B$ and $[y/z]B$. Here A is the "base space," and all B taken over every element of A is known as the "total space." B for some z is known as a fiber.

We introduce the transport tr, with the following specification:

$$x, y{:}A,\ p{:}\mathsf{Id}_A(x, y) \vdash \mathsf{tr}[z.B](p) : [x/z]B \to [y/z]B$$

Informally, this says that a transport, which is specified by a total space ($z.B$) and a path p, takes elements from the fiber associated with the start of the path to the end-of-path fiber. We can define transport to be:

$$\mathsf{tr}[z.B](p) := \mathsf{J}[m.n._.[m/z]B \to [n/z]B](p; m.\lambda w.w)$$

Because the motive is $[m/z]B \to [n/z]B$, we can use the identity function inside J as then the motive will become $[m/z]B \to [m/z]B$. Outside, however, because $p : \mathsf{Id}_A(x, y)$, this will become $[x/z]B \to [y/z]B$. The intuition here is that because the two elements x and y are "equal," we don't need anything more than a glorified identity function to transport between the induced fibers.