The Type Theory Zoo

@ionathanch

January 11, 2021

Contents

Chapter 1

Inductive Structures

1.1 How to Define Data

You can get away with a lot when you have dependent functions, dependent pairs, and an equality type. They do represent, after all, the universal quantifier, the existential quantifier, and equality in predicate logic, and using $(A : \text{Type}) \to A$ as your notion of falsehood (since it is never inhabited), you're able to express negation. What more do you need?[1]

Unfortunately, the world is not made of predicates and propositions. To talk about various properties of things, we need *things* to talk about. Historically things are made out of judgement rules directly, and these come in five different kinds:

1. **Formation rules**, which describe how to form a new type, such as Natfor the Peano naturals;
2. **Introduction rules**, which describe how to build inhabitants (if any) of your new type, which would be zeroand succ for Nat;
3. **Elimination rules**, which describe how to use the inhabitants;
4. **Computation rules**, which describe how an eliminator wrapped around a constructor uses its components; and optionally
5. **Uniqueness rules**, which describe how a constructor wrapped around an eliminator produces the same term that was eliminated.

The problem is that these rules are part of the type theory, and the programmer cannot[2] arbitrarily introduce new rules. They should be able to introduce new pieces of data to use during proving and programming, under the close scrutiny of well-typedness. So far, there are three common ways to allow this:

1. **Inductive data definitions**, where formation and introduction forms defined individually are subject to strict conditions to ensure consistency, and there is a common elimination form;
2. **W types**, where there is a single set of rules for modelling *well-founded trees*, but have various limitations, such as the dependence on function extensionality; and

[1] You may notice the absence of logical disjunction here, which cannot be fully expressed by the negation of logical conjunction in intuitionistic logic. Please stop noticing. We will fix this in due time.

[2] And should not, lest they introduce an inconsistency due to an unvetted rule, or worse, destroy the decidability of type checking!

3. **Recursive** or **μ-types**, which are somewhere in between these two, being a single set of rules like W types, but also require some strict conditions.

All of these have corresponding dual concepts for (usually) infinite data structures: coinductive data definitions, M-types, and corecursive or ν-types.

1.2 Inductive Data Definitions

The general form of an inductive data definition is as follows:

$$\begin{aligned}&\text{data } I\ \Delta_P : \Delta_I \to \text{Type where}\\ &\qquad \langle c_i : \Delta_i \to I\ \Delta_P\ \vec{e}_i \rangle_i\end{aligned}$$

Δ is a *telescope* of a variable and its type $(x_1 : A_1) \dots (x_n : A_n)$, where A_j can depend on any x_i where $i < j$. We abuse this notation in several places:

- $\Delta \to B$ means the function type $(x_1 : A_1) \to \cdots \to (x_n : A_n) \to B$;
- $f\ \Delta\ \vec{y}$ means the function application $f\ x_1 \dots x_n\ y_1 \dots y_m$;
- data $I\ \Delta$ in inductive data definitions expands to data $I\ (x_1 : A_1) \dots (x_n : A_n)$;
- $e[\Delta \mapsto \vec{a}]$ means the simultaneous substitution $e[\overrightarrow{x_i \mapsto a_i}]$, where $\vec{a} = a_1 \dots a_n$; and
- $\Gamma \vdash \vec{a} : \Delta$ means that $\Gamma \vdash a_i : A_i$ all hold.

Telescopes may be empty as well, which extends to the above notation abuse in the obvious manners.

In the data definition above, Δ_P are the *parameters* of the type I, while Δ_I are its *indices*. The inductive type I itself has type $\Delta_P \to \Delta_I \to \text{Type}$. The difference between parameters and indices is that parameters must be the same for every constructor, while indices may differ. Δ_i are the *arguments* of the constructor c_i, and a fully-applied constructor would have type $I\ \Delta_p\ \vec{e}_i$, where $\vec{e}_i$ are the indices for that particular constructor's type.

1.3 Well-Formedness of Inductive Definitions

The data definition must first be well-typed. This means that $\Delta_P \to \Delta_I \to \text{Type}$ must be a well-typed function type, and each $\Delta_i \to I\ \Delta_P\ \vec{e}_i$ must be a well-typed function type in the context where $I : \Delta_P \to \Delta_I \to \text{Type}$. These correspond to the introduction and formation rules.

$$\frac{\Gamma \vdash \Delta_P \to \Delta_I \to \text{Type}_\ell : \text{Type}_k}{\Gamma \vdash I : \Delta_P \to \Delta_I \to \text{Type}_\ell}\ \text{FORM} \qquad \frac{\Gamma \Delta_P (I : \Delta_P \to \Delta_I \to \text{Type}_\ell) \vdash \Delta_i \to I\ \Delta_P\ \vec{e}_i : \text{Type}_\ell}{\Gamma \vdash c_i : \Delta_i \to I\ \Delta_P\ \vec{e}_i}\ \text{INTRO}_i$$

1.3.1 Strict Positivity

On top of well-typedness, the definition needs to satisfy some syntactic conditions. In the arguments of a constructor, I cannot appear anywhere it likes, because this can result in proofs of $\bot$. As an example, consider the following inductive definition:

$$\begin{aligned}&\text{data B} : \text{Type where}\\ &\qquad \text{b} : (\text{B} \to \bot) \to \text{B}\end{aligned}$$

For now, we define a function that returns the argument of b rather than define its eliminator or a general way to destruct constructors.

$$\begin{aligned}&\text{getB} : \text{B} \to (\text{B} \to \bot)\\&\text{getB}\ (\text{b}\ bb) \coloneqq bb\end{aligned}$$

Then we are able to prove $\bot$ by showing that B both holds and does not hold.

$$\begin{aligned}&\text{notB} : \text{B} \to \bot\\&\text{notB}\ b \coloneqq (\text{getB}\ b)\ b\end{aligned}$$

$$\begin{aligned}&\text{falsehood} : \bot\\&\text{falsehood} \coloneqq \text{notB}\ (\text{b notB})\end{aligned}$$

The following similar inductive definition also allows us to prove $\bot$.

$$\begin{aligned}&\text{data B} : \text{Type where}\\&\quad \text{b} : (((\text{B} \to \bot) \to \bot) \to \bot) \to \text{B}\end{aligned}$$

$$\begin{aligned}&\text{getB} : \text{B} \to (((\text{B} \to \bot) \to \bot) \to \bot)\\&\text{getB}\ (\text{b}\ bb) \coloneqq bb\end{aligned}$$

$$\begin{aligned}&\text{doubleNeg} : (B : \text{Type}) \to B \to (B \to \bot) \to \bot\\&\text{doubleNeg}\ B\ b\ notB \coloneqq notB\ b\end{aligned}$$

$$\begin{aligned}&\text{tripleNeg} : (B : \text{Type}) \to (((B \to \bot) \to \bot) \to \bot) \to B \to \bot\\&\text{tripleNeg}\ B\ notnotnotB\ b \coloneqq notnotnotB\ (\text{doubleNeg}\ B\ b)\end{aligned}$$

$$\begin{aligned}&\text{notB} : \text{B} \to \bot\\&\text{notB}\ b \coloneqq \text{tripleNeg B}\ (\text{getB}\ b)\ b\end{aligned}$$

$$\begin{aligned}&\text{falsehood} : \bot\\&\text{falsehood} \coloneqq \text{notB}\ (\text{b}\ (\text{doubleNeg}\ (\text{B} \to \bot)\ \text{notB}))\end{aligned}$$

Note that doubleNeg and tripleNeg are stated in terms of a general type B. In fact, if we are allowed to define a constructor whose inductive type appears to the left of an odd number of arrows in an argument type, we are always able to prove $\bot$. Given $\text{b} : (((\text{B} \to \bot) \to \dots) \to \bot) \to \text{B}$ for an odd number of arrows, we can produce a term of type $\text{B} \to \bot$ by a repeated application of tripleNeg, a term of type B by a repeated application of doubleNeg, and finally falsehood.

Therefore, we must at the very least disallow an inductive type to appear to the left of an odd number of arrows in the type of any of its constructors' arguments, since such a constructor would effectively be a proof that falsehood of that type implies truthhood of the type. This is called the *positivity condition*, while the positions to the left of an odd number of arrows are *negative positions*, since being to the left of an odd number of $\bot$ implies simple negation.

Most also impose a *strict positivity condition* and disallow the inductive type to appear to the left of *any* number of arrows for various reasons, since a type theory with an impredicative Prop type

universe (or similar) could again derive $\bot$ using merely positive inductive definitions [12]. The key idea the latter is that a constructor of type $b : ((\mathrm{B} \to \mathrm{Type}) \to \mathrm{Type}) \to \mathrm{B}$ would enable creating an injective function $f : (\mathrm{B} \to \mathrm{Type}) \to \mathrm{B}$ through the injection $i : (\mathrm{B} \to \mathrm{Type}) \to ((\mathrm{B} \to \mathrm{Type}) \to \mathrm{Type})$. If types are viewed as sets, then the cardinality of $\mathrm{B} \to \mathrm{Type}$ is $|\mathrm{Type}|^{|\mathrm{B}|}$, but injectivity of f asserts that $|\mathrm{Type}|^{|\mathrm{B}|} \leq |\mathrm{B}|$, which is clearly a contradiction (assuming that at least two types exist). This extends to B appearing to the left of any even number of arrows, since it can always be reduced two arrows by tripleNeg.

1.3.2 Mutual Inductive Definitions

So far, inductive data definitions can implicitly depend on previously-defined ones. We now consider mutual definitions that depend on one another. To prevent an illegible proliferation of subscripts, we instead deal with two concrete mutual inductive definitions.

$$\begin{aligned}&\text{data Tree } (A : \mathrm{Type}) : \mathrm{Type} \text{ where}\\ &\quad \mathrm{Tree} : A \to \mathrm{Forest}\ A \to \mathrm{Tree}\ A\end{aligned}$$

$$\begin{aligned}&\text{data Forest } (A : \mathrm{Type}) : \mathrm{Type} \text{ where}\\ &\quad \mathrm{nil} : \mathrm{Forest}\ A\\ &\quad \mathrm{cons} : \mathrm{Tree}\ A \to \mathrm{Forest}\ A \to \mathrm{Forest}\ A\end{aligned}$$

A forest is a specialized list of trees, while a tree is some datum A along with a forest. Intuitively, along with the usual strict positivity conditions, Forest should also only appear strictly positively in the argument types of Tree and vice versa. Furthermore, all constructors tree, nil, and cons must be well-typed under the context containing both Tree and Forest. However, Tree is not to be typed under the context containing Forest, and vice versa. (On the other hand, if two inductive definitions have constructors that mutually use the other inductive type and one inductive type uses the other as an index, then these become *inductive-inductive* definitions.)

Mutual definitions can sometimes be encoded as a single inductive type with an index to distinguish between the former mutual definitions [4], especially when disregarding issues with universe levels. For instance, using elements of Bool to differentiate between trees and forests, we can define a combined TreeForest type.

$$\begin{aligned}&\text{data TreeForest } (A : \mathrm{Type}) : \mathrm{Bool} \to \mathrm{Type} \text{ where}\\ &\quad \mathrm{tree} : A \to \mathrm{TreeForest}\ A\ \mathrm{false} \to \mathrm{TreeForest}\ A\ \mathrm{true}\\ &\quad \mathrm{nil} : \mathrm{TreeForest}\ A\ \mathrm{false}\\ &\quad \mathrm{cons} : \mathrm{TreeForest}\ A\ \mathrm{true} \to \mathrm{TreeForest}\ A\ \mathrm{false} \to \mathrm{TreeForest}\ A\ \mathrm{false}\end{aligned}$$

Then we can simply redefine Tree A as TreeForest A true and Forest A as TreeForest A false.

To summarize so far, suppose that for a constructor c_i of an inductive type I we have

$$\Delta_i = (x_{i1} : A_{i1}) \to \cdots \to (x_{im} : A_{im}).$$

Then each A_{ij} must have the form $\Delta_{ij} \to D_{ij}$, and I must not appear in Δ_{ij} to satisfy strict positivity. In the case of mutual definitions for inductive types $\vec{I}$, none of $\vec{I}$ may appear in any Δ_{ij} of any constructor of any type.

1.3.3 Nested Positivity

In the constructor argument type $\Delta_{ij} \to D_{ij}$ for an inductive type I, we may have that $D_{ij} = I'\ \Delta_P\ \vec{e}_{ij}$, where I may or may not be the same inductive type as I'. The obvious question follows: could I appear in the parameters or the indices?

Consider again the mutually-defined inductives for trees and forests. Instead of redefining a custom type for a list of trees as forest, we may wish to reuse the usual list type.

$$\begin{aligned}&\text{data Tree } (A : \text{Type}) : \text{Type where}\\ &\quad \text{tree} : A \to \text{List } (\text{Tree } A) \to \text{Tree } A\end{aligned}$$

This seems like a perfectly reasonable definition to allow. (In fact, mutual inductive definitions may generally be desugared into nested inductive definitions, even when the definitions belong to different universe levels [2, Section 8.6.3].) Since Tree appears in the parameter of List, we may generally wish to allow I in $\vec{p}_{ij}$. But again, if it's allowed to appear *anywhere*, we are able to prove $\bot$:

$$\begin{aligned}&\text{data Box } (A : \text{Type}) : \text{Type where}\\ &\quad \text{box} : A \to \text{Box } A\end{aligned}$$

$$\begin{aligned}&\text{data B } : \text{Type where}\\ &\quad \text{b} : \text{Box } (\text{B} \to \bot) \to \text{B}\end{aligned}$$

We can compose the respective getBox and getB functions that return the constructor arguments to yield a function of type $\text{B} \to (\text{B} \to \bot)$, and from these three we can proceed to define notB and isB as before. So I must again appear strictly positively in $\vec{p}_{ij}$.[3] On the other hand, I cannot appear at all in the indices (or in any parameter that any constructor uses as an index), since we can prove $\bot$ by equating that index with an argument in a negative position.

$$\begin{aligned}&\text{data } (=) : \text{Type} \to \text{Type} \to \text{Type where}\\ &\quad \text{refl} : (A : \text{Type}) \to A = A\end{aligned}$$

$$\begin{aligned}&\text{data B} : \text{Type where}\\ &\quad \text{b} : (A : \text{Type}) \to A = B \to (A \to \bot) \to B\end{aligned}$$

$$\begin{aligned}&\text{notB} : \text{B} \to \bot\\ &\text{notB } (\text{b B refl } notB) \coloneqq notB\end{aligned}$$

$$\begin{aligned}&\text{falsehood} : \bot\\ &\text{falsehood} \coloneqq \text{notB } (\text{b B refl notB})\end{aligned}$$

There is a subtle issue here that prevents this definition from type checking as it is. Suppose $\text{B} : \text{Type}_\ell$ were in universe level ℓ. Then the type of b must be in at least universe level $\ell{+}1$, since it takes $A : \text{Type}_\ell$ as an argument. To circumvent this, everything can be made to exist in a impredicative Prop universe instead (which can be done in Coq), or *induction-recursion* (simultaneously defining an inductive data definition and a recursive function that use each other) can be used to "shrink" the universe of A (which can be done in Agda) [6].

To summarize yet again what we know so far, we mutually define the notions of strict and nested positivity, and state the conditions for a valid constructor type.

[3] In fact, if $I = I'$, then I can even appear *negatively* in the parameters! For instance,

$$\text{data } I \ (A : \text{Type}) : \text{Type where } c : I \ (I \ A \to \bot) \to I \ A$$

is a valid inductive data definition in Agda. It's unclear whether such a definition would be at all useful in any way.

- A constant X appears strictly positively in the type $\Delta \to D$ if I does not appear in Δ and nested positively in the type D.
- A constant X appears nested positively in the type $I\ \vec{p}\ \vec{e}$ if X appears strictly positively in each of $\vec{p}$, strictly positively in the return types of the constructors of $I\ \vec{p}\ \vec{e'}$, and not at all in $\vec{e}$.
- The constructor type $c_i : \Delta_i \to I\ \Delta_P\ \vec{e}_i$ is well-formed if I appears strictly positively in each of Δ_i and not all in $\vec{e}_i$.

Finally, let us restate the general form of a (non-mutual) inductive data definition.

$$\begin{aligned}&\text{data } I\ \Delta_P : \Delta_I \to \text{Type where}\\ &\quad \langle c_i : \langle (x_{ij} : \Delta_{ij} \to D_{ij}) \rangle_j \to I\ \Delta_P\ \vec{e}_i \rangle_i\end{aligned}$$

Again, I must not appear in any Δ_{ij} and nested positively in D_{ij}.

1.4 Recursion on Inductive Data

Now that we have our formation and introduction rules along with the syntactic conditions imposed upon then, we need some elimination and computation rules. There are a few different ways to handle an inductive term, but they must do all of the following:

- **Discriminate** the constructors from one another (in other words, it must handle all possible constructors);
- **Decompose** the constructors into their components so that each piece may be used;
- **Recur** on the components according to the type's induction principle; and
- **Terminate** on all inputs.

For programming practicality, the ways that inductive data are eliminated don't restrict recursion to occur only on the components of the constructors, but unrestricted recursion easily allows for nontermination and therefore inconsistency. There are a number of ways to restrict recursion to prevent this, but they will not be covered here.

1.4.1 Pattern Matching

Inductive data may be handled by defining recursive functions that pattern-match on constructors. For instance, the following computes the depth of a tree with two pattern-matching definitions:

$$\begin{aligned}&\text{depth} : (A : \text{Type}) \to \text{Tree}\ A \to \text{Nat}\\ &\text{depth}\ A\ (\text{tree}\ a\ \text{nil}) \coloneqq \text{succ zero}\\ &\text{depth}\ A\ (\text{tree}\ a\ (\text{cons}\ t\ ts)) \coloneqq \max\ (\text{succ}\ (\text{depth}\ A\ t))\ (\text{depth}\ A\ (\text{tree}\ a\ ts))\end{aligned}$$

Here, max is a function that takes the maximum of two Nats. Immediately we can see that the recursion is not so simple – here, a recursive call is done on a new tree containing the rest of the list of trees, which is not directly a component of the original tree. There is an interplay between decomposition and discrimination as well, especially with inductive types with indices: pattern-matching on some argument of the function can restrict what the other arguments can be (which are then called *inaccessible* or *forced* arguments), and earlier definitions can influence which constructors need to be covered in later definitions. Generally pattern-matching is very powerful but very intricate, and are usually desugared into *case trees* in proof assistants such as Agda.

1.4.2 Case Expressions

We now turn to case expressions, which handle only discrimination and decomposition. If we use case expressions without any recursion, we are still able to define simple functions, such as the following that returns the first element of a tree:

$$\begin{aligned}
&\text{get} : (A : \text{Type}) \to \text{Tree } A \to A \\
&\text{get } A\ t \coloneqq \\
&\quad \text{case } (\lambda t : \text{Tree } A.\ A)\ t \text{ of} \\
&\qquad \text{tree} \Rightarrow \lambda a : A.\ \lambda trees : \text{List } (\text{Tree } A).\ a
\end{aligned}$$

Ignoring the second function argument of the case expression for now, we see that this is similar to pattern matching: t gets decomposed into a tree with arguments a and $trees$ through the function in the single branch of the expression. However, this is much more simple than pattern-matching, since a and $trees$ are not patterns and merely bind to the constructor arguments. Below we give the typing rules for case expressions in general, assuming that the inductive type $I : \Delta_P \to \Delta_I \to \text{Type}$ and its constructors $c_i : \Delta_P \to \Delta_i \to I\ \Delta_P\ \vec{e}_i$ abstracted over its parameters exist in the environment Γ. We also use the telescopes $\Delta_P, \Delta_I, \Delta_i$ from the inductive definition.

$$\frac{\begin{array}{c}\Gamma \vdash t : I\ \vec{p}\ \vec{e} \qquad \Gamma \vdash P : \Delta_I[\Delta_P \mapsto \vec{p}] \to (x : I\ \vec{p}\ \Delta_I) \to \text{Type} \\ \Gamma \vdash d_i : \Delta_i \to P\ \vec{e}_i[\Delta_P \mapsto \vec{p}]\ (c_i\ \vec{p}\ \Delta_i)\end{array}}{\Gamma \vdash \text{case } P\ t \text{ of } \langle c_i \Rightarrow d_i \rangle_i : P\ \vec{e}\ t}\ \textsc{case}$$

The inductive datum t being discriminated and decomposed is called the *target*, while the function P that dictates the return type of the case expression is called the *motive*. There is a *branch* or *method* d_i for each of the constructors. Because everything implicitly uses (i.e. without explicit application) the parameters $\vec{p}$ of the target's type, they need to be substituted into the types of the indices Δ_I and the constructor's return type's indices $\vec{e}_i$. If inductive definitions had only indices and no parameters (which would be just as powerful, since all parameters can be encoded as indices), the typing rule (with $I : \Delta_I \to \text{Type}$ and $c_i : \Delta_i \to I\ \Delta_P\ \vec{e}_i$ in Γ) would no longer require these substitutions.

$$\frac{\begin{array}{c}\Gamma \vdash t : I\ \vec{e} \qquad \Gamma \vdash P : \Delta_I \to (x : I\ \Delta_I) \to \text{Type} \\ \Gamma \vdash d_i : \Delta_i \to P\ \vec{e}_i\ (c_i\ \Delta_i)\end{array}}{\Gamma \vdash \text{case } P\ t \text{ of } \langle c_i \Rightarrow d_i \rangle_i : P\ \vec{e}\ t}\ \textsc{case-idx}$$

Taking it one step further, if we can make the case expression *nondependent* by removing the dependency of the motive on the indices and the target. In this case, the motive can be inferred entirely and does not need to be provided.

$$\frac{\Gamma \vdash t : I\ \vec{e} \qquad \Gamma \vdash d_i : \Delta_i \to P}{\Gamma \vdash \text{case } t \text{ of } \langle c_i \Rightarrow d_i \rangle_i : P}\ \textsc{case-nondep}$$

Notice that P and d_i are all terms of function type, but do not have to syntactically be functions: they can also be terms that *reduce* to functions. We can prevent the appearance of arbitrary function-typed terms by explicitly binding variables in an alternate match expression. Again, if the motive is nondependent, then it can be inferred.

$$\frac{\begin{array}{c}\Gamma \vdash t : I\ \vec{p}\ \vec{e} \qquad \Gamma\Delta_I(x : I\ \vec{p}\ \Delta_I) \vdash P[y \mapsto \Delta_I] : \text{Type} \\ \Gamma\Delta_i \vdash d_i[\vec{z} \mapsto \Delta_i] : P\ \vec{e}[\Delta_P \mapsto \vec{p}]\ (c_i\ \vec{p}\ \Delta_i)\end{array}}{\Gamma \vdash \text{match } x.\vec{y}.P\ t \text{ with } \langle c_i\ \vec{z} \Rightarrow d_i \rangle_i : P[x \mapsto t][\vec{y} \mapsto \vec{e}]}\ \textsc{match}$$

This results in function application in the computation rule for case expressions and substitution in the computation rule for match expressions. Since they don't rely on any of the types, we can instead write them as untyped reduction rules, sometimes referred to as *ι-reduction*.

$$\text{case } P\ (c_j\ \vec{a}) \text{ of } \langle c_i \Rightarrow d_i \rangle_i \triangleright_\iota d_i\ \vec{a}$$
$$\text{match } x.\vec{y}.P\ (c_j\ \vec{a}) \text{ with } \langle c_i\ \vec{z} \Rightarrow d_i \rangle_i \triangleright_\iota d_i[\vec{z} \mapsto \vec{a}]$$

1.4.3 Fixed Points

Case and match expressions are generally accompanied by a way to define recursive functions. When the function takes multiple arguments, one of them is deemed the *decreasing argument*, and naïve termination checking would merely ensure that recursive calls occur only on components of the decreasing argument. For simplicity, we deal with functions that explicitly state the decreasing argument, starting with the *fixpoint*, which binds the function to a constant.

$$\frac{\Gamma \vdash \Delta \to (x : I\ \vec{p}\ \vec{e}) \to P : \text{Type} \qquad \Gamma\Delta(x : I\ \vec{p}\ \vec{e})(f : \Delta \to (x : I\ \vec{p}\ \vec{e}) \to P) \vdash d : P}{\Gamma \vdash \text{fixpoint } f\ \Delta\ (x : I\ \vec{p}\ \vec{e}) : P \text{ in } d : \Delta \to (x : I\ \vec{p}\ \vec{e}) \to P}\ \textsc{fixpoint}$$

The fixpoint is presented in a style similar to the match expression, where d is the body of the fixpoint function rather than a function itself. The decreasing argument is bound to x and all other arguments are bound to Δ inside of d, as well as the fixpoint f itself. As a concrete example, the recursive function double below doubles a natural, using a fixpoint function with a case expression.

```
double : Nat → Nat
double =
  fixpoint f (n : Nat) : Nat in
    case (λn : Nat. Nat) n of
      zero ⇒ zero
      succ ⇒ λm : Nat. succ (succ (f m))
```

We can also define recursion in a more traditional form using a *fixed point operator* or *combinator* that computes the least fixed point for a function that takes as argument an approximation of the fixed point.

$$\frac{\Gamma \vdash d : (f : \Delta \to (x : I\ \vec{p}\ \vec{e}) \to P) \to \Delta \to (x : I\ \vec{p}\ \vec{e}) \to P}{\Gamma \vdash \text{fix } d : \Delta \to (x : I\ \vec{p}\ \vec{e}) \to P}\ \textsc{fix}$$

The argument f in the function d represents the entire fixed point itself, while x and Δ are again the decreasing and other arguments. Notice that if let T be the type of fix d, then we could alternatively treat fix as a function on a function (hence *operator*/*combinator*) whose type is $(T \to T) \to T$. We present again as example the recursive function double, this time using the fixed point operator and a match expression.

```
double : Nat → Nat
double =
  fix (λf : Nat → Nat. λn : Nat.
        match x.Nat n with
          zero ⇒ zero
          succ m ⇒ succ (succ (f m)))
```

Although case expressions and fixed point operators have a more function-like presentation while match and fixpoint have a more pattern-like presentation, we are free to mix and match as we like. And just like case and match expressions, fixed point operators and fixpoints have untyped reduction rules that use function application and substitution, respectively. They are also referred to as *μ-reduction* rules.

$$(\text{fixpoint } f\ \Delta\ (x : I\ \vec{p}\ \vec{e}) : P \text{ in } d)\ \vec{b}\ (c\ \vec{a}) \triangleright_\mu d[f \mapsto \text{fixpoint } f\ \Delta\ (x : I\ \vec{p}\ \vec{e}) : P \text{ in } d][\Delta \mapsto \vec{b}][x \mapsto c\ \vec{a}]$$
$$(\text{fix } d)\ \vec{b}\ (c\ \vec{a}) \triangleright_\mu d\ (\text{fix } d)\ \vec{b}\ (c\ \vec{a})$$

1.4.4 Eliminators

The data definitions we have seen are called *inductive* because the elements of the inductive type are built up inductively. That is, starting with no elements at all, we look at our constructors and see what elements we can build at each step. For example, with the naturals, from no elements at all, we see that we can construct one element: zero. Then at the next step, we can construct from that one element (succ zero), and so on. If an inductive type has a parameter A, then we begin the construction with all the elements of A. Considering the usual list type parametrize over some A, we first have nil, then one step later, we have (cons a nil) for every element $a : A$, and so forth.

Often we wish to prove properties about the elements in our inductive types. In mathematics, to prove some property P, we use *induction*: If we can show that for every possible constructor c, P holding for the components of c implies that P holds for c built out of those components, then P must hold for *all* elements of the inductive type. This idea is called the *induction principle*, the *elimination principle*, or simply the *eliminator*, since it is used to eliminate a member of an inductive type into some other type. It's also called the *recursor* as it computes recursively.

In terms of types, P is exactly the motive that we have seen in case and match expressions. For some inductive type I, beginning with parameters Δ_P, we have

$$P : \Delta_I \to (x : I\ \Delta_P\ \Delta_I) \to \text{Type}.$$

This is also known as the *major premise* of the eliminator [3]. Now consider some constructor

$$c_i : \Delta_i \to I\ \Delta_P\ \vec{e}_i.$$

where $\Delta_i = \langle (x_{ij} : \Delta_{ij} \to D_{ij}) \rangle_j$ are the constructor's arguments. For the moment, suppose that I is not a nested inductive type. This means that if $D_{ij} = I_{ij}\ \vec{p}\ \vec{e}_{ij}$, then I does not appear in $\vec{p}$.

For each constructor c_i, we need the statement P_i that if P holds for all of its inductive arguments – the *induction hypotheses* – then P holds for the fully-applied c_i. These are the *minor premises* of the eliminator [3]. We will build the type step-by-step. First, we need to provide all of the arguments, and apply P to the fully-applied constructor, as well as the indices of its type.

$$P_i : \Delta_i \to ? \to P\ \vec{e}_i\ (c_i\ \Delta_i)$$

Now consider each argument x_{ij} of c_i and its type. If D_{ij} is not an inductive type or it is and $I_{ij} \neq I$, then x_{ij} is not a recursive argument, and ? is empty. Otherwise, if $I_{ij} = I$, then we need an inductive hypothesis P_{ij}. This takes all of the arguments in Δ_{ij}, and has return type P applied to a term of type $I\ \vec{p}\ \vec{e}_{ij}$. Luckily, x_{ij} gives us exactly the term we need.

$$P_{ij} : \Delta_{ij} \to P\ \vec{e}_{ij}\ (x_{ij}\ \Delta_{ij})$$

Then all we need to do is provide the minor premise with the inductive hypotheses $\Delta_{\mathbb{J}}$.

$$\mathbb{J} = \{j \mid I_{ij} = I\}$$
$$\Delta_{\mathbb{J}} = \langle (P_{ij} : \Delta_{ij} \to P\ \vec{e}_{ij}\ (x_{ij}\ \Delta_{ij})) \rangle_{j \in \mathbb{J}}$$
$$P_i : \Delta_i \to \Delta_{\mathbb{J}} \to P\ \vec{e}_i\ (c_i\ \Delta_i)$$

If $\Delta_{\mathbb{J}}$ is empty, then P_i is a *base case*; otherwise, it is a *step case* [3]. Finally, we collect together the parameters, the major premise (motive), the minor premises (base and step cases), the indices, and finally the target, and produce a return type of P applied to the indices and the target. We also summarize the various components involved in the eliminator. While we present it as a type, it can also be presented as a typing rule by stating all of the eliminator's arguments as premises and the return type as the type of the fully-applied eliminator.

$$\begin{aligned}
\text{elim}_I : \Delta_P &\to (P : \Delta_I \to (x : I\ \Delta_P\ \Delta_I) \to \text{Type}) \\
&\to \langle (P_i : \Delta_i \to \Delta_{i\mathbb{J}} \to P\ \vec{e}_i\ (c_i\ \Delta_i)) \rangle_i \\
&\to \Delta_I \to (x : I\ \Delta_P\ \Delta_I) \\
&\to P\ \Delta_I\ x \\
\textit{where } i\mathbb{J} &= \{j \mid I_{ij} = I\} \\
\Delta_{i\mathbb{J}} &= \langle (P_{ij} : \Delta_{ij} \to P\ \vec{e}_{ij}\ (x_{ij}\ \Delta_{ij})) \rangle_{j \in i\mathbb{J}} \\
D_{ij} &= I_{ij}\ \vec{p}\ \vec{e}_{ij} \textit{ or } \text{Type } (\textit{or other base type}) \\
\Delta_i &= \langle (x_{ij} : \Delta_{ij} \to D_{ij}) \rangle_j \\
c_i &: \Delta_i \to I\ \Delta_P\ \vec{e}_i \\
I &: \Delta_P \to \Delta_I \to \text{Type}
\end{aligned}$$

- Δ_P, Δ_I: Parameters and indices
- c_i, Δ_i, $\vec{e}_i$: The ith constructor, its arguments, and the indices of its inductive type
- P, P_i: Major and minor premises
- $\Delta_{\mathbb{J}}$: Inductive hypotheses
- $\vec{e}_{ij}, \Delta_{ij}$: Recursive argument indices and unnamed (arguments to the jth arguments of the ith constructor)

In terms of our discrimination–decomposition–recursion–termination model, the minor premises discriminate between the constructors, their arguments decompose the constructors, and recursion occurs in the inductive hypotheses. From the perspective of eliminators, a case expression is "just an induction principle with its inductive hypotheses chopped off" [10]. Termination is guaranteed to occur, because the recursive calls are always the same eliminator applied to the recursive arguments, as we can see in the computation rule below. However, this also means that eliminators are not as powerful or general as pattern-matching or fixpoints, where the recursive function may be called on any argument as long as it passes whatever termination checking is present.

$$\frac{\begin{array}{c}\Gamma \vdash \vec{p} : \Delta_P \qquad \Gamma \vdash P : \Delta_I \to (x : I\ \Delta_P\ \Delta_I) \to \text{Type} \qquad \Gamma \vdash P_i : \Delta_i \to \Delta_{i\mathbb{J}} \to P\ \vec{e}_i\ (c_i\ \Delta_i) \\ \Gamma \vdash \vec{e} : \Delta_I \qquad \Gamma \vdash \vec{a} : \Delta_i \qquad \Gamma \vdash a_k : \Delta_k \to I\ \vec{p}\ \vec{e}_k\end{array}}{\Gamma \vdash \text{elim}_I\ \vec{p}\ P\ \langle P_i \rangle_i\ \vec{e}\ (c_j\ \vec{a}) \equiv P_j\ \vec{a}\ \langle \lambda \Delta_k.\ \text{elim}_I\ \vec{p}\ P\ \langle P_i \rangle_i\ \vec{e}_k\ (a_k\ \Delta_k) \rangle_{k \in j\mathbb{J}} : P\ \vec{e}\ (c_j\ \vec{a})}\ \text{COMP}$$

For convenience, the typing premise for $\vec{a}$ is stated twice for recursive arguments a_k. In this case, it should return the same inductive type I with possibly different indices $\vec{e}_k$. Notice that $\vec{e}_k$ is used on the right-hand side of the computation rule, but doesn't appear on the left-hand side: this is not a rule we can state in generality as a simple untyped reduction rule.

If I *is* a nested inductive type, then all hope is lost: there is no known general way to state the induction principle for a nested inductive definition. For particularly complex ones, these take some creativity to formulate, and are generally proven from the other methods of recursion.

1.4.5 Examples

Naturals and Finite Sets

For completeness, we begin with the simplest nontrivial inductive definition: the Peano naturals. We give the eliminators slightly different names from elim_I.

$$\begin{aligned}&\text{data Nat} : \text{Type where}\\&\quad \text{zero} : \text{Nat}\\&\quad \text{succ} : \text{Nat} \to \text{Nat}\end{aligned}$$

$$\begin{aligned}&\text{nrec} : (P : \text{Nat} \to \text{Type}) \to (pz : P\ \text{zero})\\&\quad \to (ps : (n : \text{Nat}) \to P\ n \to P\ (\text{succ}\ n))\\&\quad \to (n : \text{Nat}) \to P\ n\\&\text{nrec}\ P\ pz\ ps\ \text{zero} \coloneqq pz\\&\text{nrec}\ P\ pz\ ps\ (\text{succ}\ n) \coloneqq ps\ n\ (\text{nrec}\ P\ pz\ ps\ n)\end{aligned}$$

This is the usual mathematical induction over natural numbers: we prove that some property holds for all naturals by first showing that it holds for zero, then by showing that if it holds for n it will also hold for $n + 1$. Now that we have the naturals, we are able to define the *finite sets*, which are indexed by naturals.

$$\begin{aligned}&\text{data Fin} : \text{Nat} \to \text{Type where}\\&\quad \text{fzero} : (n : \text{Nat}) \to \text{Fin}\ (\text{succ}\ n)\\&\quad \text{fsucc} : (n : \text{Nat}) \to \text{Fin}\ n \to \text{Fin}\ (\text{succ}\ n)\end{aligned}$$

$$\begin{aligned}&\text{frec} : (P : (n : \text{Nat}) \to \text{Fin}\ n \to \text{Type}) \to (pfz : (n : \text{Nat}) \to P\ (\text{succ}\ n)\ (\text{fzero}\ n))\\&\quad \to (pfs : (n : \text{Nat}) \to (f : \text{Fin}\ n) \to P\ n\ f \to P\ (\text{succ}\ n)\ (\text{fsucc}\ n\ f))\\&\quad \to (n : \text{Nat}) \to (f : \text{Fin}\ n) \to P\ n\ f\\&\text{frec}\ P\ pfz\ pfs\ (\text{succ}\ n)\ (\text{fzero}\ n) \coloneqq pfz\ (\text{succ}\ n)\\&\text{frec}\ P\ pfz\ pfs\ (\text{succ}\ n)\ (\text{fsucc}\ n\ f) \coloneqq pfs\ n\ f\ (\text{frec}\ P\ pfz\ pfs\ n\ f)\end{aligned}$$

The type Fin n represents the set of numbers no greater than n; the type has n inhabitants. For instance, there are no possible elements of type Fin zero; fzero zero is the only element of type Fin (succ zero); fzero (succ zero) and fsucc (succ zero) (fzero zero) are the only elements of type Fin (succ (succ zero)); and so on.

Lists and Vectors

Next, we move on to the simplest nontrivial parametrized inductive definition, the list, which has already appeared in our definition of trees.

$$\begin{aligned}&\text{data List}\ (A : \text{Type}) : \text{Type where}\\&\quad \text{nil} : \text{List}\ A\\&\quad \text{cons} : A \to \text{List}\ A \to \text{List}\ A\end{aligned}$$

$$\begin{aligned}&\text{lrec} : (A : \text{Type}) \to (P : \text{List}\ A \to \text{Type}) \to (pn : P\ \text{nil})\\&\quad \to (ps : A \to (l : \text{List}\ A) \to P\ l \to P\ (\text{cons}\ a\ l))\\&\quad \to (l : \text{List}\ A) \to P\ l\\&\text{lrec}\ A\ P\ pn\ ps\ \text{nil} \coloneqq pn\\&\text{lrec}\ A\ P\ pn\ ps\ (\text{cons}\ a\ l) \coloneqq ps\ a\ l\ (\text{lrec}\ A\ P\ pn\ ps\ l)\end{aligned}$$

Analogous to lists are *vectors*, which are lists whose lengths are encoded in the type as an index, and one of the simplest nontrivial inductive definitions with both a parameter and an index.

$$\begin{aligned}&\text{data Vec } (A : \text{Type}) : \text{Nat} \to \text{Type where}\\&\quad \text{vnil} : \text{Vec } A \text{ zero}\\&\quad \text{vcons} : (n : \text{Nat}) \to A \to \text{Vec } A\ n \to \text{Vec } A\ (\text{succ } n)\end{aligned}$$

$$\begin{aligned}&\text{vrec} : (A : \text{Type}) \to (P : (n : \text{Nat}) \to \text{Vec } A\ n \to \text{Type}) \to (pvn : P \text{ zero vnil})\\&\quad \to (pvc : (n : \text{Nat}) \to A \to (v : \text{Vec } A\ n) \to P\ n\ v \to P\ (\text{succ } n)\ (\text{vcons } n\ a\ v))\\&\quad \to (n : \text{Nat}) \to (v : \text{Vec } A\ n) \to P\ n\ v\\&\text{vrec } A\ P\ pvn\ pvc \text{ zero vnil} \coloneqq pvn\\&\text{vrec } A\ P\ pvn\ pvc\ (\text{succ } n)\ (\text{vcons } n\ a\ v) \coloneqq pvc\ n\ a\ v\ (\text{vrec } A\ P\ pvn\ pvc\ n\ v)\end{aligned}$$

An Eliminator for Trees

Although we don't have a general method for deriving the eliminator for nested inductive definitions, we can still analyze particular instances and derive sensible eliminators for them. Consider for instance the Tree type. The major premise is straightforwardly

$$P : \text{Tree } A \to \text{Type}.$$

As for the minor premise corresponding to the tree constructor, we begin with

$$pt : (a : A) \to (l : \text{List } (\text{Tree } A)) \to ? \to P\ (\text{tree } a\ l).$$

For the missing inductive hypothesis, we need to capture the idea of P holding for every tree in the list, implemented using pattern matching as recursive calls on the trees. So we create a new data definition that contains these proofs.

$$\begin{aligned}&\text{data PList } (A : \text{Type})(P : A \to \text{Type}) : \text{List } A \to \text{Type where}\\&\quad \text{pnil} : \text{PList } A\ P \text{ nil}\\&\quad \text{pcons} : (a : A) \to (l : \text{List } A) \to P\ a \to \text{PList } A\ P\ l \to \text{PList } A\ P\ (\text{cons } a\ l)\end{aligned}$$

The type and the definition of the eliminator for PList is left as an exercise for the reader. Then we can fill in the missing hole in our minor premise and state the entire eliminator for Tree.

$$\begin{aligned}&\text{trec} : (A : \text{Type}) \to (P : \text{Tree } A \to \text{Type})\\&\quad \to (pt : (a : A) \to (l : \text{List } (\text{Tree } A)) \to \text{PList } (\text{Tree } A)\ P\ l \to P\ (\text{tree } a\ l))\\&\quad \to (t : \text{Tree } A) \to P\ t\\&\text{trec } A\ P\ pt\ (\text{tree } a\ ts) \coloneqq pt\ a\ ts\ ?\end{aligned}$$

As expected, the eliminator reduces by an application of the minor premise. But from where do we get a PList when we only have a List? Intuitively, a PList is a list of proofs of P for each element of the list, and trec gets us proofs of P for a tree, so we need a function that maps over Lists to produce PLists. We will use pattern matching to define this function, but we could use the eliminator for lists as well.

$$\begin{aligned}&\text{pmap} : (A : \text{Type}) \to (P : A \to \text{Type}) \to (f : (a : A) \to P\ a) \to (l : \text{List } A) \to \text{PList } A\ P\ l\\&\text{pmap } A\ P\ f \text{ nil} \coloneqq \text{pnil}\\&\text{pmap } A\ P\ f\ (\text{cons } a\ l) \coloneqq \text{pcons } a\ l\ (f\ a)\ (\text{pmap } A\ P\ f\ l)\end{aligned}$$

Finally, we can state the full Tree eliminator:

$$\text{trec } A\ P\ pt\ (\text{tree } a\ ts) \coloneqq pt\ a\ ts\ (\text{pmap } (\text{Tree } A)\ P\ (\text{trec } A\ P\ pt)\ ts)$$

Equality and Indices

Although it isn't inductive and its eliminator requires no recursion, the *equality type* is an important data definition. It asserts the equality of two terms by providing only a single constructor stating reflexivity of equality: something can only be equal to itself. The definition comes in two variants, the latter due to Paulin–Mohring [7].

$$\begin{aligned}&\text{data } (=)\ (A : \text{Type}) : A \to A \to \text{Type where}\\ &\quad \text{refl} : (a : A) \to a =_A a\\ &\text{data } (=)\ (A : \text{Type})(a : A) : A \to \text{Type where}\\ &\quad \text{refl} : a =_A a\end{aligned}$$

We use the infix notation $a =_A a$ to mean $(=)\ A\ a\ a$ and occasionally omit the type A when it is clear from context. We also delay discussion of the eliminators to a later chapter. The first definition illustrates how we can always turn parameters (as in $(a : A)$ in the second definition) into indices. However, with an equality type, we can now turn any *indices* into *parameters* by asserting equality of the parameters they've become with a constructor argument representing the former index.[4] To illustrate, we can define the finite sets as follows:

$$\begin{aligned}&\text{data Fin } (n : \text{Nat}) : \text{Type where}\\ &\quad \text{fzero} : (m : \text{Nat}) \to (n = \text{succ } m) \to \text{Fin } n\\ &\quad \text{fsucc} : (m : \text{Nat}) \to (n = \text{succ } m) \to \text{Fin } m \to \text{Fin } n\end{aligned}$$

Then what was formerly fzero zero : Fin (succ zero) now becomes fzero zero (refl (succzero)) : Fin (succ zero), trivially requiring an extra proof of equality. In short, we can turn all indices into parameters so long as we have the indexed equality type as our one single inductive definition with indices.

1.5 W Types

So far, we have only seen inductive definitions whose constructor arguments are *not* functions. The simplest such inductive definition that doesn't rely on other definitions are the *well-founded trees*, also known as *W types*.

$$\begin{aligned}&\text{data W } (A : \text{Type})\ (B : A \to \text{Type}) : \text{Type where}\\ &\quad \text{sup} : (a : A) \to (w : B\ a \to \text{W}\ A\ B) \to \text{W}\ A\ B\end{aligned}$$

Occasionally the type is written as $\text{W}(x : A).B(x)$, just as function types may be written as $\Pi(x : A).B(x)$. Intuitively, when interpreted as a tree, there is a branch for each element $a : A$, and a subtree for each element in $B\ a$. As such, the constructor sup $a\ b$ is also written infixively as $a \triangleleft b$ to evoke "a picture of a tree growing rightward from a root" [9].

W types can in fact also be interpreted as an encoding for inductive definitions themselves, where the branches are constructors and the subtrees are the recursive arguments. Taking $\bot = (A : \text{Type}) \to A$[5], $\top = (A : \text{Type}) \to A \to A$, and assuming the existence of the usual Bool, we can define structures such as the naturals and binary trees.

[4] This technique is reportedly [1] called *fording* by McBride [8].

[5] An alternate definition would be the data definition

$$\text{data } \bot : \text{Type where}$$

with no constructors, and whose eliminator is then

$$\text{elim}_\bot : (P : \bot \to \text{Type}) \to (b : \bot) \to P\ b.$$

$$
\begin{aligned}
\text{Nat} &\coloneqq \text{W Bool } (\lambda b : \text{Bool. case } b \text{ of } \langle \text{false} \Rightarrow \bot, \text{true} \Rightarrow \top \rangle) \\
\text{zero} &\coloneqq \text{sup false } (\lambda x : \bot.\ x\ \text{Nat}) \\
\text{succ } n &\coloneqq \text{sup true } (\lambda x : \top.\ n)
\end{aligned}
$$

$$
\begin{aligned}
\text{BTree} &\coloneqq \text{W Bool } (\lambda b : \text{Bool. case } b \text{ of } \langle \text{false} \Rightarrow \bot, \text{true} \Rightarrow \text{Bool} \rangle) \\
\text{leaf} &\coloneqq \text{sup false } (\lambda x : \bot.\ x\ \text{BTree}) \\
\text{node } l\ r &\coloneqq \text{sup true } (\lambda b : \text{Bool. case } b \text{ of } \langle \text{false} \Rightarrow l, \text{true} \Rightarrow r \rangle)
\end{aligned}
$$

Because the second argument always needs to produce a W type, the base cases ask for a $\bot$ so that the required W type can be produced at will. In the step cases, we return the recursive arguments depending on the input. To define structures that also non-recursive arguments, we also need to assume the existence of pairs, whose types are written as $(x : A * B)$ in the dependent case and $(A * B)$ in the nondependent case.[6] Then we can define labelled binary trees as follows, sticking the label in the first argument.

$$
\begin{aligned}
\text{BTree } A &\coloneqq \text{W } (\text{Bool} * A)\ (\lambda(b, a) : (\text{Bool} * A).\ \text{case } b \text{ of } \langle \text{false} \Rightarrow \bot, \text{true} \Rightarrow \text{Bool} \rangle) \\
\text{leaf} &\coloneqq \text{sup } (\text{false}, a)\ (\lambda x : \bot.\ x\ \text{BTree}) \\
\text{node } l\ r &\coloneqq \text{sup } (\text{true}, a)\ (\lambda b : \text{Bool. case } b \text{ of } \langle \text{false} \Rightarrow l, \text{true} \Rightarrow r \rangle)
\end{aligned}
$$

To define lists (which are essentially labelled naturals) that may possibly be empty, we need the option to supply no label to the base case, which is done using a dependent pair. We use _ to omit unneeded arguments for concision.

$$
\begin{aligned}
\text{List } A \coloneqq \text{W}\ \ & (b : \text{Bool} * \text{case } b \text{ of } \langle \text{false} \Rightarrow \top, \text{true} \Rightarrow A \rangle) \\
& (\lambda(b, \_) : (\text{Bool} * \_).\ \text{case } b \text{ of } \langle \text{false} \Rightarrow \bot, \text{true} \Rightarrow \top \rangle) \\
\text{nil} &\coloneqq \text{sup } (\text{false}, \lambda A : \text{Type}.\ \lambda a : A.\ a)\ (\lambda x : \bot.\ x\ \text{List}) \\
\text{cons } a\ l &\coloneqq \text{sup } (\text{true}, a)\ (\lambda x : \top.\ l)
\end{aligned}
$$

The first immediate downside to using W types as compared to inductive definitions is that we require at the very least an existing type with two distinguishable elements (in our case, Bool) and a (dependent) pair type, whereas with inductive definitions these can be *defined as* inductive types.

1.5.1 Typing Rules

Now that we've sufficiently convinced ourselves that W types are a possible alternative to inductive definitions, we formally state their typing rules, omitting repeated premises in the computation rule. Notice that the computation rule requires no information gained solely from the typing premises, so we could also write it as a reduction rule.

[6] The corresponding inductive type for the dependent case would be

$$
\begin{aligned}
&\text{data Pair } (A : \text{Type})\ (B : A \to \text{Type}) : \text{Type where} \\
&\quad \text{pair} : (x : A) \to B\ x \to \text{Pair } A\ B
\end{aligned}
$$

and similarly for the nondependent case. As an independent typing rule, the type is usually written as $\Sigma(x : A).B(x)$.

$$\frac{\Gamma \vdash A : \text{Type} \qquad \Gamma \vdash B : A \to \text{Type}}{\Gamma \vdash \text{W}\ A\ B : \text{Type}}\ \textsc{form-W}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma(a : A) \vdash b : B\ a \to \text{W}\ A\ B}{\Gamma \vdash \text{sup}\ a\ b : \text{W}\ A\ B}\ \textsc{intro-W}$$

$$\frac{\begin{array}{c}\Gamma \vdash A : \text{Type} \qquad \Gamma \vdash B : A \to \text{Type} \qquad \Gamma \vdash P : \text{W}\ A\ B \to \text{Type} \qquad \Gamma \vdash w : \text{W}\ A\ B \\ \Gamma \vdash p : (a : A) \to (b : B\ a \to W\ A\ B) \to (pw : (x : B\ a) \to P\ (b\ x)) \to P\ (\text{sup}\ a\ b)\end{array}}{\Gamma \vdash \text{wrec}\ A\ B\ P\ p\ w : P\ w}\ \textsc{elim-W}$$

$$\frac{\ldots \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B\ a}{\Gamma \vdash \text{wrec}\ A\ B\ P\ p\ (\text{sup}\ a\ b) \equiv p\ a\ b\ (\lambda x : B\ a.\ \text{wrec}\ A\ B\ P\ p\ (b\ x)) : P\ (\text{sup}\ a\ b)}\ \textsc{comp-W}$$

1.5.2 Recovering Induction Principles

1.5.3 W Types with Explicit Base Cases

1.6 μ Types

1.7 A Note on Notation

Throughout this section, a variety of different letters are used to mean different things in different contexts. We play fast and loose with the notational conventions because we aren't discussing *one* type theory but rather the many different features a type theory might have and how they might appear. Even so, we catalogue here the primary uses of the letters and give a brief rundown of what should be fairly self-evident syntax.

- Letters in *italics* and Greek letters are metavariables, while names in roman are constants.
- A, B, C, D, P, T are types.
- a, b, d, e, p, q are terms (also referred to as expressions). p, q in particular are proofs of propositional equality, or "paths".
- c is a constructor name, and I is an inductive type name.
- f, g, x, y, z are variables in the implicit language, often f, g for functions.
- n, m are naturals in the implicit metalanguage, used as an index to denote the lengths of sequences. For instance, $\vec{a} = a_1 \dots a_n$ is a sequence of n terms.
- i, j, k are index metavariables used as an index to denote some member of a sequence, so that $1 \le i \le n$ or the like is taken implicitly. For instance, a_i is the ith member of $\vec{a}$, and c_i is the ith c of the sequence $\langle c_i \Rightarrow d_i \rangle_i = (c_1 \Rightarrow d_1) \dots (c_n \Rightarrow d_n)$. In fact, $\vec{a}$ would merely be syntactic sugar for $\langle a_i \rangle_i$.
- k, ℓ are naturals in the implicit metalanguage, used as universe levels.
- $\mathbb{I}, \mathbb{J}$ are sets of naturals.
- $=$ is used as both an equality in the metalanguage, usually for some sort of alpha-equivalent syntactic equality, and as the propositional equality type. $:=$ is a definitional equality, binding some constant to a term, possibly with pattern-matching. Finally, $\equiv$ is judgemental equality, determined by judgement rules.

Chapter 2

Notions of Equality

2.1 An Identity Crisis

> There is so much about equality.
> I feel like 99% of type theory is about equality.
>
> *@jordyd@octodon.social*

The most archetypical example demonstrating the capabilities of dependent types is perhaps the *identity type*.[1] In proof assistants such as Coq, Agda, or Idris, we may see it defined as follows:

$$\begin{aligned}&\text{data } (=) : (A : \text{Type}) \to (a, b : A) \to \text{Type where}\\ &\quad \text{refl} : (A : \text{Type}) \to (a : A) \to a =_A a\end{aligned}$$

The only constructor the identity type has is a proof of reflexivity, which states that any given term is equal to itself. This makes perfect sense, but it doesn't appear to be any way to prove equality of anything else. You carry on nonetheless, and endeavour to prove the other two properties of an equivalence relation: symmetry and transitivity.

$$\begin{aligned}&\text{sym} : (A : \text{Type}) \to (a, b : A) \to a =_A b \to b =_A a\\ &\text{sym } A\ a\ a\ \text{refl}_a \equiv \text{refl}_a\end{aligned}$$

$$\begin{aligned}&\text{trans} : (A : \text{Type}) \to (a, b, c : A) \to a =_A b \to b =_A c \to a =_A c\\ &\text{trans } A\ a\ a\ a\ \text{refl}_a\ \text{refl}_a \equiv \text{refl}_a\end{aligned}$$

This seems superfluous, almost, since matching on equalities can only yield the fact that things that are equal must be the same, so there is not much to do. And so you proceed, accumulating more and more properties of equality, eventually discovering proof by induction, and you go on your merry way. But all in all, this equality business seems like a rather simple thing. Until it isn't.

Dive deep enough and you may find yourself surrounded by people who regularly sling around technical phrases and names like *intentional* and *extensional* type theory, *definitional* and *propositional* equality (isn't there just the one equality?), *Martin-Löf* and *John Major* (clearly both very prominent type theorists, as far as you can tell from context), and *J* and *K eliminators*. Sometimes it's an axiom. Sometimes many things are axioms. But most importantly, somehow every one of these have to do with the notion of identity. It's a veritable zoo of equality.

[1] In this house we do not speak of the indexed vector.

2.2 Dependent Eliminators

You've seen the definition of the identity type. It's part of the first exhibit, the *inductive data types*. The general syntax of these definitions can be expressed as:[2]

$$\begin{gathered}\text{data } I : \Delta_I \to \text{Type where} \\ \langle c_i : \Delta_i \to I\ \vec{e}_i \rangle_i\end{gathered}$$

where $\Delta = (x_1 : A_1) \to \cdots \to (x_n : A_n)$, some sequence of type bindings, and we write $|\Delta| = \vec{x}_i$. Here, Δ_I are the arguments to the inductive type I, and each Δ_i are the arguments to the constructor c_i. Just as there are constructors, we also have destructors; we will first consider *case expressions* as they are defined in CIC, Coq's core calculus. The typing rule for case expressions is:

$$\frac{\Gamma \vdash P : \Delta_I \to (p : I\ |\Delta_I|) \to \text{Type} \qquad \Gamma \vdash p : I\ \vec{a} \qquad \Gamma \vdash d_i : \Delta_i \to P\ \vec{e}_i\ (c_i\ |\Delta_i|)}{\Gamma \vdash \text{case}_P\ p\ \text{of}\ \langle c_i \Rightarrow d_i \rangle_i : P\ \vec{a}\ p}\ \textsc{case}$$

P is aptly named the *motive*, which is the goal statement that we wish to prove, given some inductive construction and its inductive type's arguments. p is the *target*, the construction we wish to destruct. Finally, for each constructor c_i of I, we have a *branch* d_i, which takes the constructor's arguments and returns the appropriate proof of P. The syntax of the case expression should be familiar, except perhaps for P. The motive is usually inferred by the proof assistant, which is a lot easier when it doesn't depend on the inductive type arguments or the target. In fact, if we consider the *nondependent* case expression, the typing rule is simplified greatly:

$$\frac{\Gamma \vdash p : I\ \vec{a} \qquad \Gamma \vdash d_i : \Delta_i \to P}{\Gamma \vdash \text{case}\ p\ \text{of}\ \langle c_i \Rightarrow d_i \rangle_i : P}\ \textsc{case-nondep}$$

We pause here to note that so far we have three parts to our inductive type feature:

1. The **formation rules** for an inductive type I, in the first line of the data definition;
2. The **introduction rules** to *build* elements of type I, in the second line of the data definition; and
3. The **elimination rules** to *use* elements of type I, or our case rule.

These rules tell us how to make (well-typed) terms of our language. Most language features follow this pattern; for instance, with functions, we have the formation rule as the function type $\Pi(x : T).U$, the introduction rule as functions $\lambda x : T.e$, and the elimination rule as function application $f\ a$ for some function f and argument a. But we are missing one more rule, which tells us how the introduction and elimination rules fit together:

4. The **computation rule**, which tells us how destructor destructs a construction.

This is also known as the reduction or rewrite rule. For functions, this is the usual β-reduction. For inductive types, it is ι-reduction:

$$\text{case}_P\ (c_j\ \vec{b})\ \text{of}\ \langle c_i \Rightarrow d_i \rangle_i \triangleright_\iota d_j\ \vec{b}$$

This is all fine and dandy if we were using Coq, but if we were using Agda or Idris instead, we have one additional feature in our toolbox: *pattern-matching*. This allows us to define functions on inductive types by considering only those cases where a syntactic constructor is passed to the

[2] For now, we ignore the *parameters* of the data types, and consider only the *indices* Δ_I.

function. This sounds a lot like the elimination and computation rules, which begs the question: Can we define them using a pattern-matching function? The answer is *yes*, and we can do so systematically using the elimination rule. If we consider each premise of CASE as an argument and the type of the case expression as our return type, we have the following function signature (with some reordering):

$$\begin{aligned}\mathrm{elim}_I : &(P : \Delta_I \to (p : I\ |\Delta_i|) \to \mathrm{Type}) \\ &\to (d_i : \Delta_i \to P\ \vec{e}_i\ (c_i\ |\Delta_i|))_i \\ &\to \Delta_I \to (p : I\ |\Delta_I|) \\ &\to P\ |\Delta_i|\ p\end{aligned}$$

The only significant change is that we've abstracted the inductive type's arguments $\vec{a}$ as Δ_I in order to use them in both the target's type and the return type. We then implement this function by pattern-matching on the target p:

$$\mathrm{elim}_I\ P\ \vec{d}_i\ \vec{a}\ (c_j\ \vec{b}) \equiv d_j\ \vec{b}$$

This function is called the *eliminator* for inductive types, and together with pattern-matching encompasses the elimination and computation rules.

We now revisit the identity type, which is also called *Martin-Löf identity*, *propositional equality*, *homogenous equality*, or simply *equality*.

$$\begin{aligned}\mathrm{data}\ (=) : (A : \mathrm{Type}) \to (a, b : A) \to \mathrm{Type}\ \mathrm{where} \\ \mathrm{refl} : (A : \mathrm{Type}) \to (a : A) \to a =_A a\end{aligned}$$

Recall that the only constructor of the identity type is refl, which is a proof of equality reflexivity. In other words, two terms are equal if they evaluate to the same thing (which is also a notion with many names). Specializing the inductive eliminator to the identity type, we have:

$$\begin{aligned}\mathrm{elim}_= : &(P : (A : \mathrm{Type}) \to (a, b : A) \to (p : a =_A b) \to \mathrm{Type}) \\ &\to (d : (A : \mathrm{Type}) \to (a : A) \to P\ A\ a\ a\ \mathrm{refl}_a) \\ &\to (A : \mathrm{Type}) \to (a, b : A) \to (p : a =_A b) \\ &\to P\ A\ a\ b\ p \\ \mathrm{elim}_=\ &P\ d\ A\ a\ a\ \mathrm{refl}_a \equiv d\ A\ a\end{aligned}$$

Since P and d are only ever applied to A, we can move A to the beginning as our first argument. Then the eliminator takes the form of what is traditionally called the *J eliminator* for identity types.

$$\begin{aligned}\mathrm{J} : &(A : \mathrm{Type}) \to (P : (a, b : A) \to (p : a =_A b) \to \mathrm{Type}) \\ &\to (d : (a : A) \to P\ a\ a\ \mathrm{refl}_a) \\ &\to (a, b : A) \to (p : a =_A b) \\ &\to P\ a\ b\ p \\ \mathrm{J}\ &A\ P\ d\ a\ a\ \mathrm{refl}_a \equiv d\ a\end{aligned}$$

You may notice the redundancy in having two identical arguments due to refl_a enforcing that a and b be the same. And intuitively, two terms should be equal only if they are the same, so why should p be an equality across two distinct variables? As it turns out, we can define using pattern-matching a similar function called the *K eliminator* that acts on equalities of identical terms.

$$\begin{aligned}\mathrm{K} : &(A : \mathrm{Type}) \to (a : A) \to (P : (p : a =_A a) \to \mathrm{Type}) \\ &\to (d : P\ \mathrm{refl}_a) \to (p : a =_A a) \to P\ p \\ \mathrm{K}\ &A\ a\ P\ d\ \mathrm{refl}_a \equiv d\end{aligned}$$

2.3 Definitions of Equality

2.3.1 Definitional Equality

Notice that in the function definitions above, we use a different, more "fundamental" equality, denoted with $\equiv$. This point deserves repeating: the equality $\equiv$ is **not** the same as the identity type $=$. This equality is called *definitional equality*, *judgemental equality*, or *computational equality*. Many complex type systems tend to separate definitional equality into *reduction rules*, *conversion rules*, and possibly more, but since they all deal with the issue of when two terms are "the same", we will refer to all their relevant rules uniformly as definitional equality.[3] A type system will then consist of three kinds of judgement forms:

- $\vdash \Gamma$, which indicates when a context is well-formed;
- $\Gamma \vdash a : A$, which indicates when a term is well-typed; and
- $\Gamma \vdash a \equiv b : A$, which indicates when two terms are definitionally equal.

We omit the well-formedness rules and their appearance in the premises, since where they occur is a topic for another day. We assume the usual typing rules for functions. Finally, we also have the following definitional equality rules outside of those from computational rules.

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash a \equiv a : A}\ \text{REFL} \qquad \frac{\Gamma \vdash a \equiv b : A}{\Gamma \vdash b \equiv a : A}\ \text{SYM} \qquad \frac{\Gamma \vdash a \equiv b : A \quad \Gamma \vdash b \equiv c : A}{\Gamma \vdash a \equiv c : A}\ \text{TRANS} \qquad \frac{\Gamma \vdash a \equiv b : A \quad \Gamma \vdash A \equiv B : \text{Type}}{\Gamma \vdash a \equiv b : B}\ \text{CONV}$$

2.3.2 Propositional Equality

We now redefine propositional equality in terms of our typing judgements instead of using an inductive definition. Recall that we have a formation rule, an introduction rule, an elimination rule, and a computation rule[4]. The first three are typing judgements, while the last is a definitional equality rule.

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma \vdash a : A \quad \Gamma \vdash b : A}{\Gamma \vdash a =_A b : \text{Type}}\ =\text{-FORM} \qquad \frac{\Gamma \vdash A : \text{Type} \quad \Gamma \vdash a : A}{\Gamma \vdash \text{refl}_a : a =_A a}\ =\text{-INTRO}$$

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma \vdash P : (a, b : A) \to (p : a =_A b) \to \text{Type} \quad \Gamma \vdash d : (a : A) \to P\ a\ a\ \text{refl}_a \quad \Gamma \vdash a : A \quad \Gamma \vdash b : A \quad \Gamma \vdash p : a =_A b}{\Gamma \vdash \text{J}\ A\ P\ d\ a\ b\ p : P\ a\ b\ p}\ =\text{-ELIM}$$

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma \vdash P : (a, b : A) \to (p : a =_A b) \to \text{Type} \quad \Gamma \vdash d : (a : A) \to P\ a\ a\ \text{refl}_a \quad \Gamma \vdash a : A}{\Gamma \vdash \text{J}\ A\ P\ d\ a\ a\ \text{refl}_a \equiv d\ a : P\ a\ a\ \text{refl}_a}\ =\text{-COMP}$$

For completeness, we provide also the elimination and computation rules for the K eliminator. Henceforth we also assume that the arguments to the computation rules are well-typed as in the

[3] In particular, we will be ignoring *congruence rules*.

[4] The arguments to the J eliminator can be reordered so that P is abstracted only over b and d is no longer a function. This is called the *Paulin-Mohring eliminator*; see Licata [7] for further details.

elimination rule and omit identical premises with ....

$$\frac{\Gamma \vdash A : \text{Type} \qquad \Gamma \vdash a : A \qquad \Gamma \vdash P : (p : a =_A a) \to \text{Type} \qquad \Gamma \vdash d : P\ \text{refl}_a \qquad \Gamma \vdash p : a =_A a}{\Gamma \vdash \text{K}\ A\ a\ P\ d\ p : P\ p}\ =\text{-ELIM-K} \qquad \frac{\ldots}{\Gamma \vdash \text{K}\ A\ a\ P\ d\ \text{refl}_a \equiv d : P\ \text{refl}_a}\ =\text{-COMP-K}$$

The K is important because as opposed to J, we can prove the *unicity* or *uniqueness of identity proofs* (UIP), which states that all proofs of equality are equal one another. We first prove that all proofs of equality are equal to refl, which we name RIP, as the provided proof of equality is on the right side. (Correspondingly, we name the symmetric variant LIP.) Then UIP can be proven either by an application of J, or by symmetry and transitivity.

$$\begin{aligned}&\text{RIP} : (A : \text{Type}) \to (a : A) \to (q : a =_A a) \to \text{refl}_a = q\\ &\text{RIP}\ A\ a\ q \coloneqq \text{K}\ A\ a\ (\lambda q : a =_A a.\ \text{refl}_a = q)\ (\lambda a : A.\ \text{refl}_{\text{refl}_a})\ q\end{aligned}$$

$$\begin{aligned}&\text{UIP} : (A : \text{Type}) \to (a, b : A) \to (p, q : a =_A b) \to p = q\\ &\text{UIP}\ A\ a\ b\ p\ q \coloneqq \text{J}\ A\ (\lambda a, b : A.\ \lambda p : a =_A b.\ (q : a =_A b) \to p = q)\ (\lambda a : A.\ \text{RIP}\ A\ a)\ a\ b\ p\ q\end{aligned}$$

We use $\coloneqq$ for definitional equalities of functions that can be written as lambda abstractions (in this case, $\text{RIP} \equiv \lambda A.\ \lambda a.\ \lambda q.\ \ldots$) and we may omit type annotations or term subscripts for clarity.

2.3.3 Heterogenous Equality

In the previous section we mentioned that propositional equality is also called homogenous equality.[5] This is because there is a different form of propositional equality where the terms on either side of the equality may not have the same type, called *heterogenous equality* or *John Major equality*.[6] This definition is still intuitively valid, since the only constructor we have for the equality is still reflexivity.

$$\frac{\Gamma \vdash A : \text{Type} \qquad \Gamma \vdash B : \text{Type} \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash a \overset{\star}{=}^{A}_{B} b : \text{Type}}\ \overset{\star}{=}\text{-FORM} \qquad \frac{\Gamma \vdash A : \text{Type} \qquad \Gamma \vdash a : A}{\Gamma \vdash \text{refl}^{\star}_{a} : a \overset{\star}{=}^{A}_{A} a}\ \overset{\star}{=}\text{-INTRO}$$

$$\frac{\begin{array}{c}\Gamma \vdash P : (A : \text{Type}) \to (B : \text{Type}) \to (a : A) \to (b : B) \to (p : a \overset{\star}{=}^{A}_{B} b) \to \text{Type}\\ \Gamma \vdash d : (A : \text{Type}) \to (a : A) \to P\ A\ A\ a\ a\ \text{refl}^{*}_{a}\\ \Gamma \vdash A : \text{Type} \qquad \Gamma \vdash B : \text{Type} \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B \qquad \Gamma \vdash p : a \overset{\star}{=}^{A}_{B} b\end{array}}{\Gamma \vdash \text{J}^{\star}\ P\ d\ A\ B\ a\ b\ p : P\ A\ B\ a\ b\ p}\ \overset{\star}{=}\text{-ELIM}$$

$$\frac{\ldots}{\Gamma \vdash \text{J}^{\star}\ A\ P\ d\ A\ A\ a\ a\ \text{refl}^{\star}_{a} \equiv d\ a : P\ A\ A\ a\ a\ \text{refl}^{\star}_{a}}\ \overset{\star}{=}\text{-COMP}$$

An interesting property of heterogenous equality is that we can prove RIP (for $\overset{\star}{=}$, evidently) directly from $\text{J}^{\star}$.

[5] "Heterogenous" and "homogenous" are actually newer variant spellings of "heterogeneous" and "homogeneous"; we will stick to the former spelling.

[6] Unlike Martin-Löf equality, John Major equality was not created by some John Major. The name was coined by Conor McBride for obscure political British reasons because he's just 🌶 spicy 🌶 like that.

$$\text{RIP}^\star : (A, B : \text{Type}) \to (a : A) \to (b : B) \to (q : a \stackrel{\star}{=}^A_B b) \to \text{refl}_a \stackrel{\star}{=} q$$
$$\text{RIP}^\star\ A\ B\ a\ b\ q \coloneqq \text{J}^\star\ (\lambda A, B, a, b.\ \lambda q : a \stackrel{\star}{=}^A_B b.\ \text{refl}^\star_a \stackrel{\star}{=} q)\ (\lambda A, a.\ \text{refl}^\star_{\text{refl}^\star_a})\ A\ B\ a\ b\ q$$

The key is the heterogenous equality used in $\text{refl}^\star_a \stackrel{\star}{=} q$: The term on the left has type $a \stackrel{\star}{=}^A_A a$, whereas the term on the right has type $a \stackrel{\star}{=}^A_B b$. In the homogenous version of J, we would have to equate a term of type $a =_A a$ to a term of type $a =_A b$, which isn't possible with homogenous equality. In general, anything provable using J can also be proven with $\text{J}^\star$, but not the other way around.

2.3.4 Extensional Equality

So far, with all of our propositional equalities, definitional equality remains *intensional*, as opposed to *extensional*, which roughly means that everything that is provably (propositionally) equal is (definitionally) equal. An important property to note is that type checking is decidable for intensional definitional equality (which we refer to as computational equality) and undecidable for *extensional* definitional equality (which we refer to as judgemental equality).[7]

There are several ways to make definitional equality extensional. The simplest is by *equality reflection*, which turns a propositional equality into a definitional one. This is a new rule for the definitional equality judgement form, but it doesn't really correspond to a notion of "computation"; it also acts like an elimination rule since it eliminates a propositional equality in the premises, but it doesn't create a new eliminator form either. We define it below for homogenous equality, although it applies to heterogenous equality as well. We also modify the introduction rule for propositional equality so that we can freely translate between definitional and propositional equalities.

$$\frac{\Gamma \vdash A : \text{Type} \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : A \qquad \Gamma \vdash p : a =_A b}{\Gamma \vdash a \equiv b : A}\ =\text{-REFLECT} \qquad \frac{\Gamma \vdash A : \text{Type} \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : A \qquad \Gamma \vdash a \equiv b : A}{\Gamma \vdash \text{refl} : a =_A b}\ =\text{-INTRO}$$

We can see this renders type checking undecidable because checking whether two arbitrary terms a, b are definitional equal amounts to doing a proof search for propositional equality.

A second way of making definitional equality extensional is to include what is known as a *uniqueness rule* or an *η-expansion* rule. Whereas a computation rule describes how a destructor acts on a constructor, a uniqueness rule describes how a constructor builds a term from destructed components. For example, the η-expansion rule for functions states that $(\lambda x : A.\ f\ x) \equiv f$, while for pairs it states that $(\text{fst}\ x, \text{snd}\ x) \equiv x$. Since we have dependent elimination, the general formulation is:

$$\frac{\ldots \qquad \Gamma \vdash e : (a, b : A) \to (p : a =_A b) \to P\ a\ b\ p}{\Gamma \vdash \text{J}\ A\ P\ (\lambda a : A.\ e\ a\ a\ \text{refl}_a)\ a\ b\ p \equiv e\ a\ b\ p : P\ a\ b\ p}\ =\text{-UNIQ}$$

[7] Different sources assign different meanings to these terms. For instance, nLab [11] distinguishes between definitional equality, which behaves merely like our $\coloneqq$, and computational equality, which describes a unidirectional notion of evaluation. Hofmann [5], on the other hand, says:

> Summing up, we can say that definitional equality is intensional–it identifies objects which become identical upon definitional expansion and/or computation...

And on the topic of extensional equality:

> Strictly speaking, definitional equality does not become extensional here, but the equality judgement no longer expresses definitional equality. To avoid of a third notion of equality, *judgemental equality*, we shall...

Meanwhile, the HoTT book [13] uses definitional and judgemental equality interchangeably.

Using this rule, given some equality $p : a =_A b$, we can also obtain a definitional equality between a and b. We first define the following functions:

$$P : (a, b : A) \to (p : a =_A b) \to \text{Type}$$
$$P\ a\ b\ p \coloneqq A$$

$$e_l : (a, b : A) \to (p : a =_A b) \to P\ a\ b\ p$$
$$e_l\ a\ b\ p \coloneqq a$$

$$e_r : (a, b : A) \to (p : a =_A b) \to P\ a\ b\ p$$
$$e_r\ a\ b\ p \coloneqq b$$

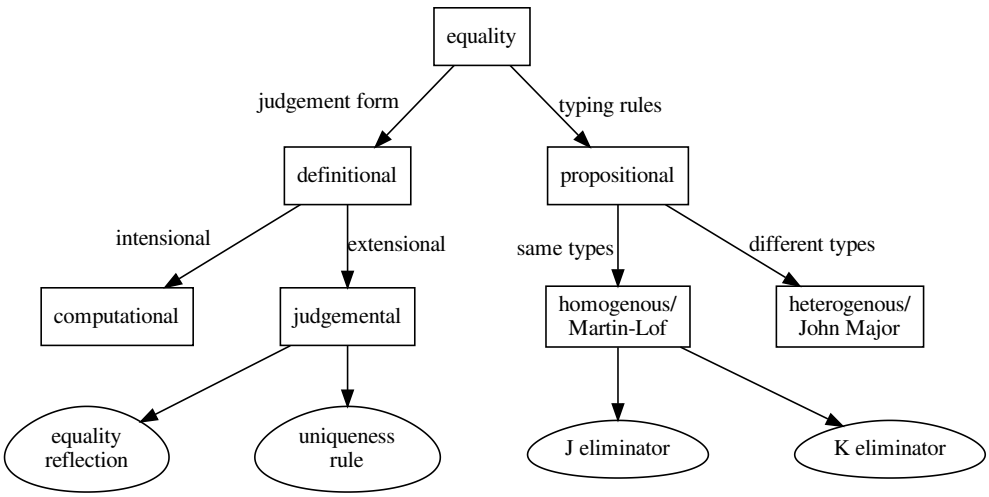
Now we can derive a definitional equality between a and b in type A up to congruence:

$$\begin{aligned}
a &\equiv e_l\ a\ b\ p && \text{(by } \beta\text{-reduction)} \\
&\equiv \mathrm{J}\ A\ P\ (\lambda a : A.\ e_l\ a\ a\ \mathrm{refl}_a)\ a\ b\ p && \text{(by =-UNIQ)} \\
&\equiv \mathrm{J}\ A\ P\ (\lambda a : A.\ a)\ a\ b\ p && \text{(by } \beta\text{-reduction)} \\
&\equiv \mathrm{J}\ A\ P\ (\lambda a : A.\ e_r\ a\ a\ \mathrm{refl}_a)\ a\ b\ p && \text{(by } \beta\text{-reduction)} \\
&\equiv e_r\ a\ b\ p && \text{(by =-UNIQ)} \\
&\equiv b && \text{(by } \beta\text{-reduction)}
\end{aligned}$$

Since there exists a derivation tree to $a \equiv b : A$ from $a =_A b$, we can see that =-UNIQ suffers from the same undecidability problem as =-REFLECT.

2.3.5 Summary

The below tree summarizes the relationships between the various definitions of equality. Again, the use of the terms "computational equality" and "judgemental" equality here are nonstandard.



2.4 Properties of Propositional Equality

There are a few well-known properties of homogenous (and therefore heterogenous) equality that we state here without proof, which can be defined using J or pattern-matching on refl. First, we reassert that equality is an equivalence relation that is symmetric and transitive as well as reflexive. If an equality between a and b is viewed as a path between points a and b in some type space A, then these are also called *inversion* and *concatenation* (or *composition*).

$$\begin{aligned}\text{sym} &: (A : \text{Type}) \to (a, b : A) \to a =_A b \to b =_A a\\ \text{trans} &: (A : \text{Type}) \to (a, b, c : A) \to a =_A b \to b =_A c \to a =_A c\end{aligned}$$

Equality also satisfies *congruence* and *substitution*. From the path perspective, these are named *ap* (since a function is *applied* to both ends) and *transport* (since you move from one endpoint to the other). We name the function application version of congruence as *happly* for homotopy reasons we will never discuss. It's worth noting that a dependent function can be applied in cong if the equality is heterogenous. The heterogenous versions happly* and subst* merely replace the homogenous equality by a heterogenous one.

$$\begin{aligned}\text{cong} &: (A, B : \text{Type}) \to (f : A \to B) \to (a, b : A) \to a =_A b \to f\ a =_B f\ b\\ \text{cong}^\star &: (A : \text{Type}) \to (B : A \to \text{Type}) \to (f : (a : A) \to B\ a) \to (a, b : A) \to a \overset{\star}{=}^A_A b \to f\ a \overset{\star}{=}^{Ba}_{Bb} f\ b\\ \text{happly} &: (A : \text{Type}) \to (B : A \to \text{Type}) \to (f, g : (a : A) \to B\ a) \to f = g \to (a : A) \to f\ a =_{Ba} g\ a\\ \text{subst} &: (A : \text{Type}) \to (P : A \to \text{Type}) \to (a, b : A) \to a =_A b \to P\ a \to P\ b\end{aligned}$$

An interesting consequence is that we can now define the heterogenous version of the K eliminator using nothing but proofs derived from J*:

$$\begin{aligned}&\text{K}^\star : (A : \text{Type}) \to (a : A) \to (P : (p : a \overset{\star}{=}^A_A a) \to \text{Type}) \to (d : P\ \text{refl}^\star_a) \to (p : a \overset{\star}{=}^A_A a) \to P\ p\\ &\text{K}^\star\ A\ a\ P\ d\ p \coloneqq \text{subst}^\star\ A\ P\ \text{refl}^\star_a\ p\ (\text{RIP}^\star\ A\ A\ a\ a\ p)\ d\end{aligned}$$

As opposed to K, which *cannot* be derived from J, K* *can* be derived from J*. All of this from merely declaring that equality may be written with two types!

2.5 Extensional Concepts

As we have seen with equality reflection, we can add various rules to the type theory to obtain different properties when it comes to definitional and propositional equality; however, that rule in particular has the nasty property of causing the undecidability of type checking. Luckily, there are other rules that are not as nasty that also bring to the type theory some notion of extensionality[8], in the set-theoretical sense we will never discuss.

2.5.1 Definitional UIP

Our current formulations of RIP define it as a function that returns a proof of propositional equality between refl_a and any $p : a =_A a$. This can be formulated as a *definitional* equality as well with a new rule (and correspondingly for heterogenous equality). Alternatively, we can instead formulate a definitional UIP rule, which is what most type theories would do, then prove RIP from UIP, but

[8] In fact, Hofmann [5] shows that adding equality reflection is a *conservative extension* of the same type theory with these other rules, in that they are both able to assign types to exactly the same terms and no more.

we stick to RIP since it's easier to use.

$$\frac{\Gamma \vdash A : \mathrm{Type} \qquad \Gamma \vdash a : A \\ \Gamma \vdash b : A \qquad \Gamma \vdash p : a =_A b}{\Gamma \vdash \mathrm{refl}_a \equiv p : a =_A b}\ \mathrm{RIP} \qquad \frac{\Gamma \vdash A : \mathrm{Type} \qquad \Gamma \vdash B : \mathrm{Type} \\ \Gamma \vdash a : A \qquad \Gamma \vdash b : B \qquad \Gamma \vdash p : a \stackrel{\star}{=}^{A}_{B} b}{\Gamma \vdash \mathrm{refl}^{\star}_a \equiv p : a \stackrel{\star}{=}^{A}_{B} b}\ \mathrm{RIP}^{\star}$$

Suppose we define subst as our eliminator for the identity type, rather than J.

$$\frac{\Gamma \vdash A : \mathrm{Type} \qquad \Gamma \vdash P : A \to \mathrm{Type} \\ \Gamma \vdash a : A \qquad \Gamma \vdash b : A \qquad \Gamma \vdash p : a =_A b \qquad \Gamma \vdash d : P\ a}{\Gamma \vdash \mathrm{subst}\ A\ P\ a\ b\ p\ d : P\ b}\ \text{=-ELIM-SUBST}$$

$$\frac{\ldots}{\Gamma \vdash \mathrm{subst}\ A\ P\ a\ a\ \mathrm{refl}_a\ d \equiv d}\ \text{=-COMP-SUBST}$$

Then we can derive both J and K from subst with a functional version of RIP.

$$\begin{aligned}
&\mathrm{RIP} : (A : \mathrm{Type}) \to (a : A) \to (p : a =_A a) \to \mathrm{refl}_a = p \\
&\mathrm{RIP}\ A\ a\ p \coloneqq \mathrm{refl}_p \qquad\qquad \text{(by the RIP rule)} \\
\\
&\mathrm{J} : (A : \mathrm{Type}) \to (P : (a, b : A) \to (p : a =_A b) \to \mathrm{Type}) \\
&\quad \to (d : (a : A) \to P\ a\ a\ \mathrm{refl}_a) \to (a, b : A) \to (p : a =_A b) \to P\ a\ b\ p \\
&\mathrm{J}\ A\ P\ d\ a\ b\ p \coloneqq \mathrm{subst}\ A\ (\lambda b : A.\ (p : a =_A b) \to P\ a\ b\ p)\ a\ b\ p\ \mathrm{Paap}\ p \\
&\quad \textit{where} \\
&\qquad \mathrm{Paap} : (p : a =_A a) \to (P\ a\ a\ p) \\
&\qquad \mathrm{Paap}\ p \coloneqq \mathrm{subst}\ (a =_A a)\ (P\ a\ a)\ \mathrm{refl}_a\ p\ (\mathrm{RIP}\ A\ a\ a\ p)\ (d\ a) \\
\\
&\mathrm{K} : (A : \mathrm{Type}) \to (a : A) \to (P : (p : a =_A a) \to \mathrm{Type}) \\
&\quad \to (d : P\ \mathrm{refl}_a) \to (p : a =_A a) \to P\ p \\
&\mathrm{K}\ A\ a\ P\ d\ p \coloneqq \mathrm{subst}\ (a =_A a)\ P\ \mathrm{refl}_a\ (\mathrm{RIP}\ A\ a\ a\ p)\ d
\end{aligned}$$

For J, the idea is that we subst over p from $P\ a\ a\ p$ to $P\ a\ b\ p$, and to obtain $P\ a\ a\ p$ we subst over RIP from $P\ a\ a\ \mathrm{refl}_a$. For K, we simply subst over RIP from $P\ \mathrm{refl}_a$ to $P\ p$.

In essence, subst plays the role of the J eliminator, while UIP (or RIP) is in a sense equivalent to the K eliminator, since one can be derived from the other. If UIP is desired in the type theory, then one may pick any one of the following combinations:

- The J and K eliminators (J implies subst and K implies UIP);
- The J eliminator and UIP (J implies subst and UIP with subst implies K);
- The subst eliminator and UIP (together imply J and K);
- The subst and K eliminators (K implies UIP and subst with UIP implies J); or
- The $\mathrm{J}^{\star}$ eliminator (implies $\mathrm{UIP}^{\star}$ and $\mathrm{subst}^{\star}$, which together imply $\mathrm{K}^{\star}$)

However, if UIP is not desired, then it is not sufficient to replace subst by J. We can also do the same for heterogenous equality, but since $\mathrm{J}^{\star}$ can derive all of $\mathrm{K}^{\star}$, $\mathrm{UIP}^{\star}$, and $\mathrm{subst}^{\star}$, one might as well assert only $\mathrm{J}^{\star}$.

2.5.2 Function Extensionality

Function extensionality states that if two functions of the same type are equal at every point in their domain, then they themselves are equal. This is perhaps the most intuitive conception of what "extensionality" means: the values of the functions at each point in the domain are their "extensions", and extensionality is the property that two things are identified solely by their extensions.

$$\frac{\Gamma \vdash f : (a : A) \to B\ a \qquad \Gamma \vdash A : \mathrm{Type} \qquad \Gamma \vdash B : A \to \mathrm{Type} \qquad \Gamma \vdash g : (a : A) \to B\ a \qquad \Gamma \vdash h : (a : A) \to f\ a =_{Ba} g\ a}{\Gamma \vdash \mathrm{funext}\ A\ B\ f\ g\ h : f = g}\ \textsc{funext}$$

Unfortunately, there is no computation rule for functional extensionality, which means that any computation will get "stuck" on its uses. There are some other concepts that can provide computational meaning to functional extensionality, such as *cubical type theories* (via path types), or *higher inductive types* (via an interval type).

On the other hand, functional extensionality can be derived using equality reflection and the uniqueness rule (η-expansion) for functions. Specifically, with some mild fudging:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash f : (a : A) \to B\ a \qquad \Gamma \vdash A : \mathrm{Type} \qquad \Gamma \vdash B : A \to \mathrm{Type} \qquad \Gamma \vdash g : (a : A) \to B\ a \qquad \Gamma \vdash h : (a : A) \to f\ a =_{Ba} g\ a}{\Gamma, (a : A) \vdash h\ a : f\ a =_{Ba} g\ a}\ \textsc{reflect}}{\Gamma, (a : A) \vdash f\ a \equiv g\ a : B\ a}\ \lambda\textsc{-intro}}{\Gamma \vdash \lambda a : A.\ f\ a \equiv \lambda a : A.\ g\ a : B\ a}\ \lambda\textsc{-uniq}}{\Gamma \vdash f \equiv g : (a : A) \to B\ a}\ \textsc{=-form}}{\Gamma \vdash \mathrm{refl} : f = g}\ \lambda\textsc{-elim}$$

2.5.3 Setoids and Quotient Types

> Setoid and 0-groupoid are from category theory. 0-types / isSet types are from HoTT. Bishop sets are a particular way to obtain set quotients constructively in CZF [Constructive Zermelo–Fraenkel set theory]. Quotient types are the type theory analogue of set quotients, a specific case of setoid types / isSet types.
>
> — *@jordyd@octodon.social*

Setoids?? 0-groupoids?? Bishop sets?? Quotient types?? I don't understand these??? what

References

[1] Guillaume Allais, Edwin Brady, Ohad Kammar, and Jeremy Yallop. "Frex: indexing modulo equations with free extensions". In: *Type-Driven Development*. 2020. URL: https://icfp20.sigplan.org/details/tyde-2020-papers/1/Frex-indexing-modulo-equations-with-free-extensions-Extended-Abstract-.

[2] Bruno Barras. "Semantical Investigations in Intuitionistic Set Theory and Type Theories with Inductive Families". Habilitation thesis. Université Paris Diderot (Paris 7), July 2012. URL: http://www.lsv.fr/~barras/habilitation/barras-habilitation.pdf.

[3] Peter Dybjer. "Inductive families". In: *Formal Aspects of Computing*. Vol. 6. July 1994, pp. 440–465. DOI: 10.1007/BF01211308. URL: http://www.cse.chalmers.se/~peterd/papers/Inductive_Families.ps.

[4] Hugo Herbelin and Arnaud Spiwack. "The Rooster and the Syntactic Bracket". In: *19th International Conference on Types for Proofs and Programs (TYPES 2013)*. Ed. by Ralph Matthes and Aleksy Schubert. Vol. 26. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014, pp. 169–187. ISBN: 978-3-939897-72-9. DOI: 10.4230/LIPIcs.TYPES.2013.169. URL: http://drops.dagstuhl.de/opus/volltexte/2014/4631.

[5] Martin Hofmann. "Extensional concepts in intensional type theory". PhD thesis. University of Edinburgh, July 1995. URL: http://www.lfcs.inf.ed.ac.uk/reports/95/ECS-LFCS-95-327/.

[6] ionathanch. *Intuition behind nested positivity and counterexamples*. URL: https://cstheory.stackexchange.com/q/48145.

[7] Dan Licata. *Just Kidding: Understanding Identity Elimination in Homotopy Type Theory*. Apr. 2011. URL: https://homotopytypetheory.org/2011/04/10/just-kidding-understanding-identity-elimination-in-homotopy-type-theory/.

[8] Conor McBride. "Dependently Typed Functions and their Proofs". PhD thesis. University of Edinburgh, July 2000. URL: https://era.ed.ac.uk/handle/1842/374.

[9] Conor McBride. *W-types: good news and bad news*. Mar. 2010. URL: https://mazzo.li/epilogue/index.html%3Fp=324.html.

[10] Conor McBride, Healfdene Goguen, and James McKinna. "A Few Constructions on Constructors". In: *Types for Proofs and Programs*. Ed. by Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 186–200. ISBN: 978-3-540-31429-5. URL: http://www.e-pig.org/downloads/concon.pdf.

[11] nLab. *equality*. Jan. 2021. URL: http://ncatlab.org/nlab/show/equality.

[12] Vilhelm Sjöberg. *Why must inductive types be strictly positive?* Apr. 2015. URL: http://vilhelms.github.io/posts/why-must-inductive-types-be-strictly-positive/.

[13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013.

Appendix A

A.1 Functions

$$\frac{\Gamma \vdash A : \text{Type} \qquad \Gamma, (x : A) \vdash B : \text{Type}}{\Gamma \vdash (x : A) \to B : \text{Type}}\ \lambda\text{-FORM} \qquad \frac{\Gamma \vdash A : \text{Type} \qquad \Gamma, (x : A) \vdash b : B}{\Gamma \vdash \lambda x : A.\ b : (x : A) \to B}\ \lambda\text{-INTRO}$$

$$\frac{\Gamma \vdash f : (x : A) \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B[x \mapsto a]}\ \lambda\text{-ELIM}$$

$$\frac{\Gamma \vdash A : \text{Type} \qquad \Gamma \vdash a : A \qquad \Gamma(x : A) \vdash B : \text{Type} \qquad \Gamma, (x : A) \vdash b : B}{\Gamma \vdash (\lambda x : A.\ b)\ a \equiv b[x \mapsto a] : B[x \mapsto a]}\ \lambda\text{-COMP}$$

$$\frac{\Gamma \vdash A : \text{Type} \qquad \Gamma(x : A) \vdash B : \text{Type} \qquad \Gamma \vdash f : (x : A) \to B}{\Gamma \vdash (\lambda y : A.\ f\ y) \equiv f : (x : A) \to B}\ \lambda\text{-UNIQ}$$