Notions and Notations in Type Theory

Eric Demko

January 5, 2021

Abstract

There are a lot of notions in type theory (even excluding dependent type theory and its descendants), and even more notations. I'm not sure which notions play well with each other, which can be synthesized, or which are the most ergonomic notations, so I'm compiling them all. As a side-effect, I'll also have a compilation of type-theoretic concepts as a reference guide to the literature, and the LaTeX source will serve as a reference for typesetting.

Most Important Next Steps

1. Nominal types; it might have something to so with [1]§3.4.
2. Higher Inductive Types
3. Indexed W-types
4. Presenting a Formal System or Programming Language (I hear that elimination, computation, and identity rules are derivable from formation and introduction [1])

I've chosen to give rules for typing $\mathsf{ind}\ f$ where f is the "body" of the induction because that's the form of the actual recursive function we're interested in. Supplying all the arguments to the recursive function gives too many premises. Supplying not even the body function (and any implicit parameters) create too large a type in the conclusion.

Contents

Part I
General and Metatheoretic Notation

1 Classes of Formal System

Formal type theories are always **formal systems** or **system** for short, so anything can be described as a "system". TODO: **theory**, **calculus**, **machine**

TODO: I keep using "type theory" for "dependently-type total functional programming language"

2 Metavariables

Metavariables (**metavar** for short) are names given to the objects of the theory, and are therefore meta-theoretic (part of the—usually informal—meta-language). The base letter chosen for a metavariable generally determines what sort of theoretic things can appear in its place. Conventions for which letters correspond to which things vary wildly between theories, since the objects of such theories vary significantly. One thing that does seem well-agreed-on is that the base letter can be modified with subscripts and/or primes (i.e. from base letter e to metavariables e, e', e'', e_i, $e_{i,j}$, e_{ij}, and e'_i, among others).

2.1 Multiple Metavariables

Often, there is a need for a list (or non-empty list, set, or other container) of metavariables.

$\overline{e}$	very basic, and easy to typeset
$\overline{e_i}$	give an implicit index variable i to each, likely to refer to individual metavariables later
	TODO: I forgot how I've typeset harpoons over long expressions, if I ever have

Often, it is understood that there must be at least one metavar, but just as often there could be zero. By the same token, it could be that some metavariables are identical (so the theoretic objects must match), but it's often understood that the metavariables are distinct. I haven't seen (or don't remember) any text tries to answer these questions with explicit notation; a reasonable reader is meant to figure it out without effort.

2.2 Metavariable Comprehensions

When I give formal descriptions of real languages, I often find myself needing quite complex sets of related metavariables. Thus, I've extended the usual overline notation to "**metavariable comprehensions**". I haven't seen any discussion in the literature (please write me if you have seen it!), so I'll go over it in some detail.

Definition 1. If $\mathscr{I}$ is an index set and ϕ is a formula involving metavariables, then we write $\overline{\phi}^{i\in\mathscr{I}}$ where i is a meta-meta-variable ranging over $\mathscr{I}$ and bound in ϕ so that each metavariable in ϕ may be indexed by $\mathscr{I}$ by mentioning i in its name. Where no confusion will result, the index set may be left implicit, and so can the binder that introduces the meta-meta-variable.

That's a lot of garbage, so let's see some examples—it can't get any more informal than that last bit of prose, but hopefully the idea is clearer than the jargon makes it appear. Let's say we have n functions $\overline{g_i}$ and matching arguments $\overline{a_i}$, then

$$f\ \overline{(g_i\ a_i)}^{i\in\mathbb{N}_n}$$

is an expression that calls each function on its corresponding argument, then passes all those results to another function f. When I see i as a metavariable, I expect it to range over some naturals or integers, so I expect readers will understand if I write $f\ \overline{(g_i\ a_i)}^{i}$, and since there's only one meta-meta-variable, I'll likely write just

$$f\ \overline{(g_i\ a_i)}$$

Once metavariable comprehensions become nested (it's rare, but possible), including the meta-meta-variable binders is useful:

$$\mathsf{let}\,\overline{x_i = f_i\ \overline{a_{ij}}^{j\in\mathbb{N}_i}}^{i}\ \mathsf{in}\ e'$$

takes a triangle of arguments, applies each row to a (possibly different) function, and binds each result to a different variable. It's contrived to be sure, but just in case I get carried away with something less contrived, it's nice to have a notation that uses fewer ellipses than

$$\mathsf{let}\,x_1 = f_1; x_2 = f_2\ a_{2,1}; \ldots; x_n = f_n\ a_1\ a_2\ \ldots\ a_{n,n-1}\ \mathsf{in}\ e'$$

Again, I've left notation ambiguous as regards the question of which kind of data structure is being comprehended over, and simply hope confusion will not result. The question is even more complex here, because there are so many more ways that metavariables could relate to each other. Most commonly, this shows up in substitutions: is $\overline{x_i \mapsto e_i}$ allowed to map the same variable twice (an ordered finite map), or not (just a simple finite map)? Again, I expect these details won't be difficult to accidentally fill in as part of the prose description. Nevertheless, it'd be nice to specify it explicitly so that computer implementation can become easier.

It can happen that the overline for metavar comprehensions is typeset lower than is aesthetic. In such cases a `\mathstrut` can help. There's probably also some way to insert a zero-width box with a given height for cases where `\mathstrut` is too tall, I just don't know the command off-hand.

2.3 Specifying Constraints on Free Variables in the Metavariables

This seems to be more common in mathematical logic than in type theory. In general, type theorists are willing to write $\lambda x.\, e\ x \longrightarrow e$ with a side-condition $x \notin \mathrm{fv}(e)$. This can get tedious sometimes: we might want to use a metavariable that stands for terms that don't have free variables other than those in the implied context around the whole expression and those explicitly mentioned. For this, I've really only seen one notation:

$e(x, y, z)$	the only variables allowed to be free in e are x,y,z, and the variables of some implicit context
$e[x, y, z]$	again, but from Dyber about inductive sets and families

This notation is ambiguous with some application notations, but it doesn't usually matter. We can also think of e having some unnamed slots into which the mentioned variables are substituted, just as application substitutes values into (possibly named) slots. Interestingly, I've yet to see this notation taken advantage of to write η-conversion without the side-condition: $\lambda x.e()\ x \longrightarrow e$, perhaps because it just looks too strange. It's a shame, because the notation could be quite useful—perhaps if the list of variables were superscripted it would be more palatable?

$$\lambda x.\, e^{()}\ x \quad \longrightarrow \quad e$$

The flip side of constraining free variables is introducing bound variables. These notations have a long lineage, but I'm not sure if they are strictly metavariable-related. Perhaps the idea is that, since variable binding is a matter of surface syntax, we need not name the binders in the theory—though it does mean that binding is now metatheoretical. Regardless, authors seem to find these notations convenient enough to use them fairly commonly.

$x.e$	x is fresh (in an assumed context) and bound in e
$[x]e$	equivalent notation from (TODO: cite) DIY Type Theory

This allows us to think of a λ as a unary constructor, ranging over expressions binding one variable: $\lambda(x.e)$ or $\lambda([x]e)$ rather than a binary constructor taking a variable and an expression separately $\lambda x.e$. The difference for λ is fortuitous in its typographic subtlety—usually the parenthesis are dropped and the two become indistinguishable!—but the notation really shines when introducing more complex binding forms, such as binary coproduct elimination:

$$\mathsf{ind}_{+}(\xi, x.e'_l, x.e'_r, \mathbf{inl}(a)) :\equiv e'_l(\mathbf{inl}(a)) : \xi(\mathbf{inl}(a))$$

When an author wants to be very explicit, they might even write

$$\mathsf{ind}_{+}(\xi, x.e'_l(x), x.e'_r(x), \mathbf{inl}(a)) :\equiv e'_l(\mathbf{inl}(a)) : \xi(\mathbf{inl}(a))$$

to make it clear that the only variables introduced are the ones which are bound (so $x.e(x)$ is parsed like $x.(e(x))$).

You might notice in the last example that $e'_l(\mathbf{inl}(a))$ omits the variable binding, which is normal, but possibly unintuitive. It seems that the "x." part is not part of the metavariable name, but just a prefix restricting the rage of the metavariable; whenever the same name is used elsewhere, the same restriction applies, so it is not usually repeated.

Another oddity from the last example is that the x. prefix is used in multiple places, but authors do not usually mean that each x need be filled with the same variable. It seems the variable binding sometimes extends to the meta-theory as well as the theory: when the same metavar occurs in multiple variable binding prefixes, they need not match. One clear exception is when translating one syntax to another, where presumably the real variables should at least be uniformly renamed after translation.

3 TODO: Axioms and Proofs

TODO: Gentzen proofs vs. Fitch proofs. Brouwer's notation is just Fitch in disguise, but thankfully he does use nested numbering, which makes it easier to refer to a subproof (under hypothesis).

4 TODO: Substitution

Substitution uses a multitude of forms, but they're all just maps:

$\{\overline{x \mapsto t}\}$	from wiki
$[\overline{t/x}]$	HoTT; wiki also mentions this in a note
$[\overline{t/x}]$	HoTT; wiki also mentions this in a note
$[\overline{t_i}/\overline{x_i}]$	
$[\overline{x_i} \mapsto \overline{t_i}]$	

The substitution might be seen as a total map on the set of variables, or a finite map from variables (i.e. partial), but the two are isomorphic.

Often substitutions will be introduced as mapping only a single variable, then **parallel substitutions** are defined based on them. Presumably, this means that **series substitutions** are a thing, though I've yet to see terminology for the fact that $[x \mapsto a][y \mapsto b]t = [x, y \mapsto a, [x \mapsto a]b]t$, but the fact rarely comes up.

Some authors introduce **capture-avoiding substitution** (a.k.a. **hygienic substitution**) as a correction of naïve **non-hygienic substitution**, which is useful for students to avoid a subtly wrong definition, but in practice only hygienic substitution is used, even if non-hygienic is sometimes called substitution *simpliciter*.

Frankly, the variety of notations is a disaster, especially since so many just swap the order willy-nilly. I refuse to use notations which use a slash: which side is which? I much prefer arrow-like notations, which at least give the direction of the replacement. Given that := is an old-school notation for mutable assignment, I prefer $\mapsto$.

One can also apply a substitution prefix or postfix:

θt	TODO where?
$t[t'/x]$	HoTT, wiki in both lambda calculus and logic
$t[t' := x]$	wiki on lambda calculi, Barendregdt

I like to think of substitutions as functions not only from variable to term, but also from term to term: the substitution operation merely lifts one function to another. So I'll write it prefix with the understanding that it's dropping parens from $\theta(t)$, which is abuse of names from $\mathrm{subst}(\theta)(t)$.

Another "fun" thing authors do is to specify the free variables of a term like $t(x, y, z)$ and then show the same term again, but with different expressions in those slots $t(a, b, c)$ to produce substitutions. Thus, we might write β-reduction as $(\lambda x.\, e(x))\ e' \longrightarrow e(e')$. This is especially prevalent in logical texts. For me, it's a bit noisy, can be ambiguous if there isn't a clear defining copy of the metavar, can get confused with simple application, and—most importantly—you don't get to treat substitutions as their own, separately-manipulable mathematical objects.

TODO is there such a thing as inverse substitution, or replacing *terms* by terms?

5 Sameness

TODO: judgmental equality, propositional equality, definitional equality, congruence $\cong$ or equivalence $\equiv$ under a relation, abbreviations, identity type, isomorphism

5.1 TODO: Abbreviations

TODO I'm just remembering these; I need references

- $A \coloneqq B$
- $A :\equiv B$
- $A \equiv B$ [4]
- $A \overset{\text{def}}{=} B$
- $A =_{\text{def}} B$ [4] attributes this to Burali-Forti
- $A \stackrel{\scriptscriptstyle\text{def}}{=} B$: it' a complex LaTeX expression, but looks way better than not adjusting the sizes
- $A \overset{\triangle}{=} B$
- $A \stackrel{\scriptscriptstyle\triangle}{=} B$: again, complex but aesthetic

Part II
Type Theory Notation

6 TODO: Judgments

7 Universes

The **terms** of a type theory correspond to terms of the untyped lambda calculus, and are classified by types. However, the types of a dependent type theory must also be classified, and **universes** are a very common way to do this. Universes represent a special—but very useful—case of what Pure Type Systems call **sorts**.

$$\mathsf{U}_0, \mathsf{U}_1, \ldots$$
$$\mathcal{U}_0, \mathcal{U}_1, \ldots$$
$$U \quad [4]$$

I don't remember seeing it, but I wouldn't be surprised if someone decided to start from $\mathcal{U}_1$ rather than $\mathcal{U}_0$.

TODO: is this really all that universes are about? Martin-Löf[4] describes universes as allowing the definition of more than just finite types; these new types that are added are called transfinite types

Very often the level is exceedingly boring, and can be mechanically reconstructed, so it is omitted. This style is called **typical ambiguity**, and can lead to valid-looking formulae which appear to show contradictions Nevertheless, if the levels of the universes are not reconstructible in an ambiguous term, the formula is invalid to begin with.

$$\mathsf{U}$$
$$\mathcal{U}$$

7.1 Cumulative hierarchy

In many type theories, the following judgment holds:

$$\frac{\Gamma \vdash A : \mathcal{U}_\ell}{\Gamma \vdash A : \mathcal{U}_{\ell+1}}$$

When this is the case, the theory is said to have a **cumulative hierarchy** of universes.

7.2 Type-in-type

If the judgment

$$\frac{}{\vdash \mathcal{U} : \mathcal{U}}$$

holds, then the theory is said to have **type in type**. Theoretically, this is of little interest as it leads to the usual paradoxes of set theory, but programming languages unconcerned with consistency (i.e. Haskell) often allow it. In this case, typical ambiguity is no longer a 'sort inference' problem , since the entire hierarchy collapses in to the universe which includes itself.

7.3 Special universes

Given the prevalence of polymorphic functions, the most common universe to see in type theory is $\mathcal{U}_0$, which is the universe of **small types**, or in System F_ω just "types". Indeed saying "A is a type" is a prose expression for the typing judgment $A : \mathcal{U}_0$ when working primarily with non-dependent types. In the usual lambda cube formulations, this universe is referred to as the **sort of types**.

$\mathcal{U}_0$	dependent type theory oriented notation
$\star$	lambda cube oriented notation
$*$	from Indexed Containers (but also really just a typographical variant of the last)

When working in a higher-kinded calculus (and especially one with kind polymorphism), another common universe is $\mathcal{U}_1$. Again, in the usual lambda cube formulations, this universe is referred to as the **sort of kinds**.

$\mathcal{U}_1$	dependent type theory oriented notation
$\square$	lambda cube oriented notation

TODO: I've also seen Set, Prop mentioned, almost as if they were just $\mathcal{U}_0$,$\mathcal{U}_1$. How do they actually work (I think they're drawn from CIC, but they certainly show up in Coq).

8 Functions

Function types at least have a standard notation in mathematics, even if some authors want to mix it up:

$A \to B$	standard notation; associates to the right
B^A	to emphasize the connection with category theory
$B \leftarrow A$	In Ogde de Moor's "Algebra of Programming", which is admittedly convenient for function composition

The standard notation for defining functions even looks like you'd expect in an ergonomic type theory:

$$\begin{aligned} f &: \mathbb{R} \to \mathbb{Z} \\ f(x) &= \lfloor x/2 \rfloor \end{aligned}$$

8.1 Abstractions

Often called **functions** at the term level, or also called **lambdas** after the concrete syntax.

$\lambda x.\, e$	Curry-style lambdas omit a type annotation on the variable
$\lambda x.e$	but no space is easier to type manually
$\lambda x : \sigma.\, e$	Church-style often uses a colon when types can be complex
$\lambda x{:}\sigma.\, e$	but omitting the spaces can help the reader identify precedence between all the beeps and boops of written type theory
$\lambda x^{\sigma}.\, e$	Church-style using superscript annotation makes the type less commanding, but can also scrunch up detail
$\Lambda x.\, e$	in non-dependent theories, type abstraction is often done with a capital lambda; most theories can also infer the kind, so they leave it implicit

Strictly, whether a system is Church- or Curry-style has more to do with the definitional approach taken for the calculus in question (from [6] §9.6). In Curry-style, we define terms, then semantics, and only then typing given to reject some terms. In Church-style, typing is given prior to semantics so that we need not consider ill-typed terms when given behavior. In practice, Church-style systems often annotate abstractions with the type(s) of their argument(s), whereas Curry-style systems do not. Thus, we'll see Church-style sometimes used as a synonym for manifestly-typed and Curry-style as a synonym for implicitly-typed.

The eliminator for abstractions is **application**, which in programming is called **function call**.

fe	can lead to some confusion if multi-letter variables or metavariables are allowed
$f\ e$	disambiguates that issue, but easily be missed at a glance
$f(e)$	crushes ambiguity under its heel, but at the cost of line noise
$\mathsf{Ap}(f, e)$	because $f(e)$ is notation roughly for substitution in [4]

Multiple parameters can be allowed, often as syntactic sugar for a curried version of the function:

$\lambda x, y, z.\, e$	
$\lambda x : \tau, y : \tau, z : \sigma.\, e$	combine the annotation for arguments of the same type
$\lambda x, y : \tau, z : \sigma.\, e$	combine the annotation for arguments of the same type
$\lambda x, y{:}\tau, z{:}\sigma.\, e$	but doesn't look so good when the colon has no elbow room
$\lambda x^{\tau}, y^{\tau}, z^{\sigma}.\, e$	
$\lambda x, y^{\tau}, z^{\sigma}.\, e$	combining annotations doesn't look as good when using superscripts

as can multiple arguments:

$f\ a\ b\ c$	space simply associates to the left
$f(a, b, c)$	when using parens, use standard math notation

It should be noted that a **parameter** is part of the definition of a function. An **argument** is a term that is substituted for a function's parameter name in the function's body. In practice, few people are pedantic enough to correct anyone except possibly in academic writing.

8.2 Dependent Function Types

The defining feature of dependent type theories—arguably the only type theories worthy of the name "type theory" *simpliciter*—is a generalization of non-dependent function type to the **dependent function type**, a.k.a. **Π-type**. Sometimes authors refer to this as the **cartesian product type**[2, 4], which we will see in §9 can be confusing. There are a multitude of notations for these types:

$\prod_{(x:A)} B$	quite common, but I find it not evocative of functions
$\prod_{(x\in A)} B$	gambino-hyland 2004, and likely others
$\prod(x{:}A).\, B$	a variant that doesn't direct attention away from the argument type
$\displaystyle\prod_{(x:A)} B$	suitable for display math
$\prod(A, B)$	where B is responsible for introducing a variable on its own, as in [4]
$\prod x{:}A.\, B$	the parens everywhere can be too much line noise
$\prod x^A.\, B$	or we can superscript the type, not unlike $\lambda x^\tau.\, e$
$(x : A) \to B$	stresses the function-ness; used often in actual theorem provers/programming languages; dropping the parens certainly would confuse me, but I haven't seen it either
$x^A \to B$	a (my) variant on the last notation, gives more attention to the variable binder, but could be confused for category-theoretic exponential notation
$A \,{}_x\!\!\to B$	another (my) variant which focuses on the types rather than the bound variable
${}^{x:}A \to B$	the last one can get the presubscript confused with a post-subscript on the argument type, but may need some kerning help

When using Π-related notation, curried parameters can be combined:

$\prod_{x:A,y:B} C$	
$\prod_{\substack{x:A\\y:B}} C$	when there are too many types
$(x : A) \to (y : B) \to C$	associates to the right just like non-dependent functions
$(x : A)(y : b) \to C$	found in [3]

8.3 TODO: Polymorphic Functions

$$\forall \alpha.\, \tau$$
$$\forall \alpha^{\kappa}.\, \tau$$
$$\forall \alpha{:}\kappa.\, \tau$$

That these notations all follow the same patterns as λ is not surprising. In dependent type theories with universes, you might say:

$$\forall \alpha.\, \tau \triangleq \prod \alpha^{\mathcal{U}}.\, \tau$$

8.4 Implicit Arguments and Parameters

In several languages, the elaborator infers arguments to certain functions, the types of which explain that the parameter is implicit.

$\{x : A\} \to B$	dependent function type with implicit parameter in Adga
$C\ x \Rightarrow \tau^{(x)}$	(non-dependent) typeclass argument in Haskell
$\tau^{(x)}$	implicit type arguments implicitly given from the free type variables (see next paragraph)

In this way, "obvious" or less-salient arguments can be omitted at the call site of values of these types. Regardless, it seems that the literature is much more interested in examining type theory kernels rather than formalizing their user-facing syntax. In the core theory implicit arguments are it's no different from (possibly dependent) function types; this notation is only used to drive the elaborator.

In many strongly-typed functional languages, type variables are separate from type names, so it's easy to implicitly quantify over unbound type variables. In Haskell or ML this doesn't feel like much of an implicit argument, but once we enter dependent type theory, such automatic quantification is translated into implicit parameters, and then filled with true arguments in the core.

Implicit arguments turn out to be incredibly useful, though, since function composition in a dependent theory would otherwise look like the unwieldy:

$$\circ : \prod_{A,B,C:\mathcal{U}} (B \to C) \to (A \to B) \to A \to C$$

$$\circ(\mathbb{N}, \mathbb{N}, \mathbb{N}, \mathrm{succ}, \mathrm{succ}) = \lambda x^{\mathbb{N}}.\, \mathrm{succ}\ (\mathrm{succ}\ x)$$

which is just a bit much. When it's easy (as in this case) to infer the arguments, I much prefer:

$$\begin{aligned} \_\circ\_ &: \{A, B, C : \mathcal{U}\} \to (B \to C) \to (A \to B) \to A \to C \\ \text{succ} \circ \text{succ} &= \lambda x^{\mathbb{N}}.\, \text{succ}\ (\text{succ}\ x) \end{aligned}$$

On the other hand, I find that the literature can often elide arguments that aren't so easy to figure out, which can hinder beginners. [7] is a clear offender in my book: not only is the enthusiastic programmer learning about new classes of types, homotopy theory, and the relation between them, but they also have to infer a bunch of arguments the purpose of which isn't quite yet grokked? Sure. `/rant` Admittedly, it's written for a more mathematical audience. It's certainly a matter of judgment as to which arguments to make implicit.

Indeed, even theorem provers can have trouble elaborating implicit arguments. As such, languages often have syntax for explicitly giving otherwise implicit arguments.

$e\ @\tau$	Haskell with `TypeApplications`; a bit line-noisy sometimes
e_τ	[7]; doesn't draw the eye, but it can interfere with metavar notation
$e\ \{\tau\}$	Agda (TODO Idris too I think)

These syntaxes also need ways to give only some of the implicit arguments explicitly.

$e\ @\_\ @\tau$	Haskell
$e\ \{\_\}\ \{\tau\}$	Agda
$e\ \{y = \tau\}$	Agda for dependent arguments, which has a clear advantage over positional systems

Creating a lambda with implicit arguments can also be done, though I've only seen it in Agda, not in the literature.

$\lambda\{x\}.\, e$	Agda

I'm tempted to riff on the $\Lambda x.\, e$ that we see in System F (which doesn't require an annotation because these type arguments could only be of kind $\star$, or even in System F_ω their kind is always inferrable), but I can see how it could be confusing. If so, I could also use $\forall a : A.\, B$ in place of $\{a : A\} \to B$

8.5 Chef's Choice

We'll see a lot of dependent functions from here on, but I'd rather not have the reader *i*) learn all the notations just in case, or *ii*) get too comfortable with only one notation. And indeed, the different notations emphasize different parts of the function type: is the variable important?, is the type more important?, should the function-ness be emphasized?, or the polymorphism?

I will use the following notations and the associated stylesheet provides commands for them:

`\deparr{x}{A} B`	$(x : A) \to B$	Makes the function-ness salient, and also because it's a very common and standard notation
`\vardeparr[\!]{x}{A} B`	$\overset{x:}{} A \to B$	Makes the parameter name less salient (it could probably even be guessed)
`\Pitype{x}{A} B`	$\prod_{x : A} B$	Focuses on the parameterization, but doesn't prioritize either the variable or its type
`\Pitypes{x:A\\y:B\\z:C} T`	$\prod_{\substack{x:A\\y:B\\z:C}} T$	Variant for many arguments
`\all{x^A} B`	$\forall x^A.\, B$	When it's most like polymorphism: i.e. getting the variable in scope is the most important thing. When the type is obvious, I will even elide the superscript type annotation

While I'm here, both `\Pitype` and `\Pitypes` have a starred variant which uses `\mathclap` on its argument, which can be useful in display math.

TODO: am I going to have any notation for implicit arguments? Perhaps $\forall$ is always implicit, wrap $\prod$ arguments in braces, and have a `\deparri` to use braces in place of parens. The weirdest might be $\{\overset{x:}{} A\} \to B$ or $\overset{x:}{}\{A\} \to B$, but hopefully I can use an always-implicit-$\forall$ in that case. If I don't want $\forall$ to be always implicit, I could have $\forall\{x^A\}.\, B$, but I'm not sure I like it; perhaps there's something I could use for overriding a default-implicit-$\forall$ to be explicit?

TODO: specifying explicit arguments. I'm thinking $\mathit{compose}_{\{A:\tau, C:\tau'\}}$, but when named arguments are too onerous $\mathit{compose}_{\{\tau,\_,\tau'\}}$ (assuming $\mathit{compose} : \forall A, B, C.\, (B \to C) \to (A \to B) \to A \to C$). The thing is, should I really insist on the braces when the base function is clearly not a metavariable? Braces for the implicit parameter of the identity type would just be annoying: let $\cdot = \cdot : \forall A^{\mathcal{U}}.\, A \to A \to \mathcal{U}$, then $x = y \rightsquigarrow x =_{\{t\}} y$ just looks worse than $x =_t y$. Perhaps it's simple subscripts $\mathit{compose}_{\tau,\_,\tau'}$ when the implicit args are easy, but @-applications when they're larger $\mathit{compose}\ _{@}\{C{:}\, \prod_{x:\mathbb{R}} x \leq a + x \leq b\}$. I want to keep the at-sign because curly braces might also end up used for record and variant expressions. I mean, I might use $\langle a, b \rangle$ for tuples/records, but I dunno what to do about sums/variants (which btw might need to have a result type hidden away somewhere as in $_{\Sigma}\{l{:}\, \star\}_{\{l:\mathbb{1}, r:\mathbb{N}\}}$).

9 TODO: Product and Sum Types

Although many languages have some sort of finite product types (e.g. structures, tuples, records), few have their duals. Since category theory has names for most of these, I've defaulted to using category-theoretic terminology in this section, but I will call out alternate terminologies as normal.

For the record, I'm a big fan of when a calculus already includes one concept, it also includes the dual. As it happens, since universal and existential quantification are logical duals, I'd include dependent products (analogous to $\exists$) along with dependent function (analogous to $\forall$).

9.1 TODO: Binary product type

TODO: simple binary product

- $\mathsf{pr}_1, \mathsf{pr}_2$
- $\mathbf{outl}, \mathbf{outr}$
- π_1, π_2

[4] calls this the **disjoin union** of a family of sets. I mean, it does make sense: it looks like an indexed family of sets, which can be thought of as a disjoint union, but what are we gonna call $A + B$? It turns out [4] calls them the **disjoint union** of *two sets* (emphasis mine), which is true ($A+B \overset{\text{def}}{=} \sum_{x:2} \mathsf{rec}_2(A, B, \mathcal{U})$), but IMO too concrete. Regardless, Martin-Löf also mentions more "traditional" notations $\coprod_{x \in A} B_x$ $\bigcup_{x \in A} B_x$

9.2 TODO: Dependent product type

- $\Sigma a : A.B$
- $\Sigma a^A.B$
- $\Sigma_{a:A} B$
- $(a : A) \times B$
- $A\ {}_a\!\times B$
- $\{x \in A \mid B(x)\}$ a more applied notation emphasizing its use as "all A's where B holds" ([4] writes it $\{x \in A : B(x)\}$ juscuz)

TODO: I've found $(e_1, x.e_2)$ binding x to e_1 in e_2 is quite useful for reducing duplication, and I conjecture could be used to give a hint to type inference of existential types.

9.3 TODO: Binary coproduct type

- l, r
- $\mathbf{inl}, \mathbf{inr}$
- ι_1, ι_2

9.4 TODO: Nullary product and sum types

$$\mathbb{0}, \mathbf{0}$$

$$\mathbb{1}, \mathbf{1}, \star$$

$$\star, (), 0_{\mathbb{1}}$$

9.5 TODO: Finite products and sums types

9.6 TODO: Records and Variants

i.e. label each of the terms in a product or coproduct

It's fairly easy for non-dependent products, but dependent products mean the map from label to expression is ordered. Annoyingly, real languages (Agda, Idris) allow dependent records.

TODO: simple enums are just variants of the form $\{\overline{\ell_i : \mathbb{1}}\}$

Aside on Bicartesian Closed Categories

The concepts in §§8–9 are a good match for what programmers call **structural types**. The operations used to create these types also have an analogue in category theory. In particular, a **Cartesian closed category** (**CCC**) is one which has a terminal object (corresponding to the unit type), binary products (corresponding to pair types), and exponentials (corresponding to function types). A **bicartesian closed category** (**BCCC**) is a CCC which also contains the dual concepts: an initial object (empty type) and binary coproducts (binary sum type). If a CCC is also **locally Cartesian closed**, then the analogues of dependent function and pair types are also present.

TODO: I've gotten this info off of wiki. Also, n-Category Café has some good links to the major papers and tutorials in the literature.

10 TODO: Recursion and Induction

10.1 TODO: Recursive types

TODO: μ- and ν-types

10.2 Inductive Types

As I understand it, μ- and ν-types can allow for the definition of some "meaningless" types like $\mu x.(x \to x)$, which would (TODO: I think) undermine a total functional language. Nevertheless, without some form of inductive definition, total languages would be restricted to only finite types (i.e. we wouldn't even be able to define the natural numbers!). **Inductive types** describe **well-founded trees**, and are how type theory is able to access (potential) infinity.

$\mathsf{W}_{(a:A)} B$	[7]
$\mathop{\mathsf{W}}\limits_{(a:A)} B$	variant analogous to those for Π-types
$\mathsf{W}(a:A).\,B$	variant analogous to those for Π-types
$\mathsf{W}(A,B)$	[5]

In addition to the sans-serif font, Martin-Löf (TODO: cite) uses W (in prose at least), and (TODO: cite Wellfounded Trees and Dependent Polynomial Functors) uses $\mathcal{W}$. Values of W-type are introduced with a single constructor `sup`, though whether the arguments are passed in parens or curried matches the surrounding notation.

In strongly-typed general-purpose functional languages like ML or Haskell, the question of recursively-defined types is not so involved. In the definition of an abstract data type, we are simply allowed to refer to the type being defined, and even other types defined later in a module, which allows the same paradoxes as μ− and ν-types. Because W-types have to be correct by construction (which I like about them anyway!), they have to carve up familiar concepts in a different way.

Recall that an ADT is defined as the sum of products; each term of the sum is named with a **constructor**, and each factor in the associated product—each **argument** of the constructor—might refer only to previously-defined types—might be a **non-inductive argument**—or it might refer to the type being defined—might be an **inductive argument**. In order for the type to make sense, each inductive argument should only refer to the type under definition in a strictly positive sense (TODO: fact check).

In a W-type $\mathsf{W}_{x:L}\, A$, we have a label type L, and an arity type A which may depend on the specific label x. The **label** type L combines the constructors and non-inductive arguments together; thus, we don't directly see the constructors. If we were thinking only in terms of ADTs, we might assume that L is an n-ary sum of products (where each factor is a non-inductive argument) for n constructors, but W-types are more general (we'll return to this later).

The **arity** type represents the inductive argument, but it does not do so directly; instead, each non-inductive argument is "named/indexed/pointed to" by an element of A. If I were being extraordinarily clear for readers coming from ML or Haskell, I might refer to the arity as the **inductive arity**. (TODO: cite that nLab uses this terminology in its introduction on inductive types). The reason for this indirection is more clear when we consider the introduction form for W-types is $\mathsf{sup}\ a\ f : \mathsf{W}_{x:L}\, A$, where $a : L$ is the label and $f : A\ a \to \mathsf{W}_{x:L}\, A$ is a function which, given a name of an inductive argument (which must be valid for the label) returns the sub-tree placed at that name.

In fact, I didn't use "indirection" indiscriminately just now: since W-types have potentially infinite elements, we cannot just allocate the maximum possible memory that an inductive value might use—that would require infinite bits. Instead, recursive values are laid out in memory using pointers; note that when defining recursive types in C, the compiler forces you to use pointers by complaining that the type under definition is "opaque" i.e. does not (yet) have a known layout. Here, we see that $\mathsf{sup}\ a\ f$ can be laid out in finite space as well because f can just be a function/closure pointer.[a] In the simplest case, where there are only finitely many inductive arguments, f can simply be a function that indexes into an array of the inductive arguments. However, W-types can be thought to have infinite inductive arguments, as can ADTs; e.g. `data T = Z | L (Nat -> T)` which in the language of W-types is $\mathsf{W}_{x:2}\,\mathsf{case}\,x\,\mathsf{of}\,\{0 \Rightarrow \mathbb{0}; 1 \Rightarrow \mathbb{N}\}$. TODO: cpdt-book calls such types (i.e. one of the constructor args is a function returning the type under definition) **reflexive**.

Note however that W-types are well more general than ADTs in two ways. First, they allow an arbitrary type as L rather than just a finite sum of products. Second, the arity A is allowed to vary not just on some finite set type, but also on any part of the possibly infinite L; i.e. not just on the constructor name, but also on the non-inductive arguments. More formally, every ADT with n constructors can be translated to a W-type of the form $\mathsf{W}(x : \sum_{c:\mathbb{N}_n} T).\ \mathsf{let}\,x = \mathsf{pr}_1\ x\,\mathsf{in}\,A$, where $\mathbb{N}_n$ is the set of naturals $\{i \in \mathbb{N} \mid i < n\}$, and T is the set of non-inductive arguments (in indexed W-types, we'll see why I chose T), and x is shadowed in A so that A can only refer to the constructor name part of the label type.

[a]Or at least, it can be if a also needs only finite memory. Nevertheless, a is either a member of a finite type, or a previously-defined inductive type which, by induction, can be laid out in finite memory

There are other sets of names for label, arity, arguments, and so on based on visualizing values of W-type as (finite-depth) trees. Each **tree** of type $\mathsf{W}_{x:L}\,A$ is canonically a **node**, $\mathsf{sup}\ s\ c$. We can think of a node as colored by the label s—yes, the terminology gets confusing when crossing traditions like this. In graph theory "label" usually refers to extra information attached to edges, whereas "color" attaches to nodes. Some authors use the word **shape** for the node coloring/label type L; this is nicely unambiguous, thus why we have selected the metavar s above. Each node may also have a set of **children**, each one accessible via a named arc. The **names** of the arcs are drawn from the arity type $A\ s$ associated with that node's shape (we use "name" rather than "label" so as not to confuse graph-theory and type-theory terms). The fact the arc names are determined by the shape gives another good reason to use "shape": it determines not just an *internal* color, but also the *external* interface which is just what sorts of child sets are allowed. The children themselves are accessible from $\mathsf{sup}\ s\ c$ using c, which can be thought of as a **child accessor** function. If i is the name of an arc, then $c\ i$ is the target node of that arc, or in other words, the i^{th} child of $\mathsf{sup}\ s\ c$. Since $A\ s$ is often a finite set, we can also use the word **position** for the arc names, or even **index** (thus the i choice of metavar above) of the arc. Using "index" can lead to a false sense that the children form a simple list, whereas complex W-types might be better thought of as having more complex data structures for nodes' children. Finally, if we're coming from a set-theoretic perspective—thinking of W-type elements as well-founded sets—a child can also be called a **predecessor**.

Since this is the place in type theory where (dependent) recursion rears its twisty head, it's worth going over the rules for W-types in some detail. I often find it useful to consider the formation as $\mathsf{W}S.\,A$, and require $A : S \to \mathcal{U}$ be an S-indexed family. That way, I can write $\mathbb{N} \overset{\text{def}}{=} \mathsf{W}\mathbb{2}.\,\mathsf{ind}_{\mathbb{2}}(\mathbb{0}, \mathbb{1})$, so that's the formalization I'll go with. Nevertheless, I think $\mathsf{W}x^S.\,A \triangleq \mathsf{W}S.\,\lambda x^S.\,A$ is more ergonomic than point-free programming sometimes.

Formation at least is straightforward:

$$\Gamma \vdash \frac{S : \mathcal{U} \qquad A : S \to \mathcal{U}}{\mathsf{W} S.\, A : \mathcal{U}} \tag{W-FORM}$$

And introduction is not bad if you read s as the shape of the introduced node, and c as the child-accessor function (so if $i : A\ s$, we can write $c\ i$ for the i^{th} child).

$$\Gamma \vdash \frac{s : S \qquad c : A\ s \to \mathsf{W} S.\, A}{\mathsf{sup}\ s\ c : \mathsf{W} S.\, A} \tag{W-INTRO}$$

Elimination (the induction principle) is a mess, but it helps to first see the computation rule. The reduction passes the shape and child-accessor as if sup were a mere tuple $\langle s, c\rangle$, but then it adds on a third argument hyp, which is an accessor function for the inductive hypotheses. The i^{th} hypothesis $hyp\ i$ is computed as you would expect: perform the same induction on the i^{th} child $c\ i$. (FIXME: where does the A come from? Of course, it's not strictly necessary after type erasure. OTOH, I could use ind as a constant for all the induction principles, but have it accepts a type as its first argument as a sort of ad-hoc polymorphism: $\mathsf{ind}\ @(\mathsf{W}S.A)\ f\ (\mathsf{sup}\ s\ c) \ \twoheadrightarrow\ f\ s\ c\ (\lambda i^{A\ s}.\, \mathsf{ind}\ @(\mathsf{W}S.A)\ f\ (c\ i))$. There might even be some interaction with quantitative type theory)

$$\mathsf{ind}_{\mathsf{W}}\ f\ (\mathsf{sup}\ s\ c) \ \twoheadrightarrow\ f\ s\ c\ hyp\ \mathsf{where}\ hyp\ i^{A\ s} = \mathsf{ind}_{\mathsf{W}}\ f\ (c\ i) \tag{W-COMP}$$

From this, we can infer how $\mathsf{ind}_{\mathsf{W}}$ is typed, but the notation is still quite crunchy. Given a property ξ on well-founded trees, we can show $\xi\ t$ for an arbitrary tree t (equivalently fold over the tree) as long as we can supply a function f of the right form; understanding the type of f is the intricate part. Let's start with the result type of f. We see that $\mathsf{sup}\ s\ c$ is an arbitrary tree, and that f must produce the desired property ξ for that tree. The constraints on the first two arguments of f (the dependent ones) are just the axioms of W-INTRO so that $\mathsf{sup}\ s\ c$ is well-formed. However, we also want to give f access to the induction hypothesis for each child. This, we add a (dependent) hypothesis-accessor parameter of type $(i : A\ s) \to \xi\ (c\ i)$, which takes a child index $i : A\ s$ to the induction hypothesis for that child (which has type $\xi\ (c\ i)$).

$$\Gamma \vdash \frac{\xi : (\mathsf{W} S.\, A) \to \mathcal{U} \qquad f : \prod_{\substack{s:S \\ c:A\ s \to \mathsf{W} S.\, A}} ((i : A\ s) \to \xi\ (c\ i)) \to \xi\ (\mathsf{sup}\ s\ c)}{\mathsf{ind}_{\mathsf{W}}\ f : \prod_{t:\mathsf{W} S.\, A} \xi\ t} \tag{W-ELIM}$$

The corresponding recursion principle is an easier starting place to understand (and probably more-often used in programming). To derive the typing for $\mathsf{rec}_{\mathsf{W}}$ we use a constant family $\xi \mapsto \lambda\_^{\mathsf{W} S.\, A}.\, \zeta$ and simplify:

$$\Gamma \vdash \frac{\zeta : \mathcal{U} \qquad f : (s : S) \to (A\ s \to \mathsf{W} S.\, A) \to (A\ s \to \zeta) \to \zeta}{\mathsf{rec}_{\mathsf{W}}\ f : (\mathsf{W} S.\, A) \to \zeta}$$

Now we can clearly see the three parameters of f, where the dependent typing is mostly about the node shape s of the tree that f operates on. TODO: identity rule?

TODO how about `data Rose a where { Br :: a -> List (Rose a) -> Rose a }`? Obviously, it'd be easier to use $\forall a. \mathsf{W}_{a \times \mathbb{N}}. \pi_2$—which encodes a rose made of a length-tagged vector—but what if I *really* want to re-use the list type, or if there's a case where there isn't an easy isomorphism back to a length-indexed version of the type?

10.3 TODO: Coinductive types

10.4 TODO: Mutually Inductive Types

If you, like me, are a programming language enthusiast, you might see a glaring gap in what is possible with inductive types so far: how does one define a type for the syntax of a programming language? For simple languages, like the untyped lambda calculus, we can use W-types, but once there are multiple non-terminals in the grammar, W-types offer no clear implementation path. It will turn out that W-types with identity types (§11) will be sufficient (though it does seem like there are some metatheoretic cobwebs that might not be suited to your philosophy).

Nevertheless, it is still useful to describe **indexed W-types**, which describe **mutually inductive type families**, or **mutually inductive types** for short. I think (TODO) these correspond in set theory to **indexed families of sets**. The Agda stdlib mentions W-types are also called **Petersson-Synek trees**, as does a Coq library; presumably after [5], which is a good introduction for those coming from programming.

Unfortunately, there are few sources in the literature that explicitly give the deduction rules for indexed W-types. Additionally, there seem to be several ways to formulate these types (even [5] gives a variant introduction rule).

$\mathsf{Tree}(A, B, C, d, a) : \mathcal{U}$	from [5]
$\mathsf{IW}^{o,r}_{A,B} : I \to \mathcal{U}$	from Appendix A of [3]
$\mathsf{WI}_J\ S\ P^J$	sort-of? TODO cite jcont.pdf version

The difference Peterson-Synek's presentation [5] and Kaposi-Taumer's [3], is that in [5] the shapes depend on the index, whereas in [3] the shape determines the index. TODO: I think I'm partial to shape depending on index. Consider `data T (n :: Nat) where {Zero :: T n; Fin n -> T n}` if index were determined from shape, it'd be like having a different `Zero` constructor for each index `n`. In contrast, when the shape is determined by index, we can simply have the `Zero` shape appear at every index.

10.4.1 After Peterson and Synek

TODO: It seems strange to me that $\mathsf{tree}(a, b, c)$ should retain keep the index type a in the indexed W-type's canonical form. As far as I'm aware, Haskell does not do this, and I expect that type would be erased in Agda, Coq, and so on. Instead, I find their $\mathsf{tree}'(b, c)$ constructor closer in intent to real languages, and is the formalism I will proceed with. This does mean that the induction principle needs to be supplied with the index type of the start node, but this is perhaps not so strange, since the single principle encodes a mutually inductive function: we can see supplying the index type as selecting which of the functions to start from. Perhaps more strangely, [5] require the arity type to also be supplied to the induction; I'm not sure why this is necessary since it can be derived from the type of the tree argument. Now that I think about it, so can the index type; what's actually going on here?

TODO: Actually, including the index type in the WI constructor basically selects which type is being constructed in the same way as supplying it to the induction principle specifies which of the mutually-recursive functions to start from.

TODO: S and A as for W-types. I is the index type. The required index of the children are given by r.

$$\mathsf{W}^I_r S.\, A$$

$$\Gamma \vdash \frac{I : \mathcal{U} \qquad S : I \to \mathcal{U} \qquad A : \prod_{\alpha : I} S\ \alpha \to \mathcal{U} \qquad r : \prod_{\alpha : I,\, s : S\ \alpha} A\ \alpha\ s \to I}{\mathsf{W}^I_r S.\, A : I \to \mathcal{U}} \tag{WI-FORM}$$

$$\Gamma \vdash \frac{s : S\ \alpha \qquad c : (i : A\ \alpha\ s) \to (\mathsf{WI}^I_r S.\, A)\ (r\ \alpha\ s\ i)}{\mathsf{sup}(s, c) : (\mathsf{WI}^I_r S.\, A)\ \alpha} \tag{WI-INTRO}$$

(FIXME: where does A, r come from in the result? See the similar discussion for W-COMP. Here, I need α passed explicitly—essentially selecting which of the mutually-recursive functions to call)

$$\mathsf{ind}_{\mathsf{WI}}\ f\ \alpha\ (\mathsf{sup}\ s\ c) \ \longrightarrow\ f\ \alpha\ s\ c\ (\lambda i^{A\ \alpha\ s}.\, \mathsf{ind}_{\mathsf{WI}}\ f\ (r\ \alpha\ s\ i)\ (c\ i)) \tag{WI-COMP}$$

$$\Gamma \vdash \frac{\xi : \prod_{\alpha : I} (\mathsf{W}^I_r S.\, L)\ \alpha \to \mathcal{U} \qquad f : \prod_{\substack{\alpha : I, s : S\ \alpha \\ c : (i : A\ \alpha\ s) \to (\mathsf{W}^I_r S.\, L)\ (r\ \alpha\ s\ i)}} ((i : A\ \alpha\ s) \to \xi\ (r\ \alpha\ s\ i)\ (c\ i)) \to \xi\ \alpha\ (\mathsf{sup}\ s\ c)}{\mathsf{ind}_{\mathsf{WI}}\ f : \prod_{t : (\mathsf{W}^I_r S.\, L)\ \alpha} \xi\ \alpha\ t} \tag{WI-ELIM}$$

10.4.2 After Kaposi and Raumer

Kaposi and Raumer start their Appendix A with "We recall the notion of an indexed W-type...[4]", but their reference doesn't actually present indexed W-types explicitly... maybe. I found a paper of the same name and authors published a year before which gives an Adga formalism, but even there, the presentation is very different. I haven't been able to determine if Kaposi and Raumer made up their notation. Although "recall" is perhaps the wrong word when presented with a broken reference, the fraction of the formalism they do give looks reasonable.

10.4.3 TODO: Notes on indexed W-types

Since there are so many metavars involved, let's break it down per notation:

inductive family	$\mathsf{IW}^{o,r}_{A,B}\, I$	$\mathsf{Tree}(A,B,C,d,a)$
index type	I	A
label type/shape	A	$x : A \vdash B(x)$
inductive arity type/position	$x : A \vdash B(x)$	$x : A, y : B(x) \vdash C(x,y)$
index of a node	$a : A \vdash o\ a$	a
index required of a child	$a : A, b : B\ a \vdash r\ a\ b$	$x : A, y : B(x), z : C(x,y) \vdash d(x,y,z)$

Formation:

$$\frac{I : \mathcal{U} \qquad A : \mathcal{U} \qquad B : A \to \mathcal{U} \qquad o : A \to I \qquad r : (a : A) \to B\ a \to I}{\mathsf{IW}^{o,r}_{A,B}\, I : \mathcal{U}}$$

$$\frac{A : \mathcal{U} \qquad B : A \to \mathcal{U} \qquad C : (x : A) \to B(x) \to \mathcal{U} \qquad d : (x : A) \to (y : B(x)) \to C(x,y) \to A \qquad a : A}{\mathsf{Tree}(A,B,C,d,a) : \mathcal{U}}$$

Introduction:

$$\frac{a : A \qquad c : (b : B\ a) \to \mathsf{IW}^{o,r}_{A,B}(r\ a\ b)}{\mathsf{sup}\ a\ c : \mathsf{IW}^{o,r}_{A,B}(o\ a)}$$

$$\frac{a : A \qquad b : B(a) \qquad c : (z : C(a,b)) \to \mathsf{Tree}(A,B,C,d,d(a,b,z))}{\mathsf{sup}(b,c) : \mathsf{Tree}(A,B,C,d,a)}\ (\S5\text{ variant})$$

The type is

$$\mathsf{Tree}(A,B,C,d,a)$$

where, thinking like context-free grammars, A is the set of non-terminals, $B(x)$ is the set of generating rules for $x : A$, $C(x,y)$ is the set of names for the positions of non-terminals in rule $y : B(x)$, $d(x,y,z)$ is the non-terminal symbol appearing in position $z : C(x,y)$, and a the start symbol. [5] A programmer more often works with the type operator $\mathcal{T}(a) \overset{\text{def}}{=} \mathsf{Tree}(A,B,C,d)$, since langs specify A,B,C,d in a roundabout (but much more intuitive) syntax. To construct one of these, we write $\mathsf{tree}(a,b,c) : \mathcal{T}(a)$, where b,c are, as in W-types, the label and inductive arguments (and a obvs the index of the result type).

$$\Gamma \vdash \frac{R : \forall x^A.\, \mathcal{T}(x) \to \mathcal{U} \qquad a : A \qquad t : \mathcal{T}(a) \qquad f : \prod_{\substack{x:A, y:B(x) \\ z:(i:C(x,y))\to\mathcal{T}(a') \\ h:(i:C(x,y))\to R(a',z(i))}} R(x, \mathsf{tree}(x,y,z))}{\mathsf{ind}_{\mathsf{Tree}}(t,f) : R(a,t)} \quad \begin{array}{l}\text{where} \\ \mathcal{T} \triangleq \mathsf{Tree}_{A,B,C,d} \\ \text{and } a' \triangleq d(x,y,i)\end{array}$$

$$\mathsf{ind}_{\mathsf{Tree}}(\mathsf{tree}(a,b,c),f) \;\; \longrightarrow \;\; f(a,b,c,\lambda i.\, \mathsf{ind}_{\mathsf{Tree}}(c(i),f))$$

[5] also gives a variant of $\mathsf{tree}(a, b, c)$ which omits a (the index of the inductive family) from the constructor. It's probably a more realistic model of mutually recursive types in that sense, but then the eliminator (induction principle) has to be supplied with more info about what type of tree is being eliminated.

I just wanna point out that indexed W-typed are more general than CFGs: CFGs are limited to a finite number of non-terminals and a finite number of rules, but indexed W-types can take e.g. $A = \mathbb{N}$ for infinite non-terminals or $B(a) = \mathbb{N}$ to give infinite rules for non-terminal a. Heck, even the right-hand side of a rule can have infinite non-terminals if we take $C(a, b) = \mathbb{N}$. Indeed, [5] mentions that indexed W-types could be used to "represent the context-sensitive parts of a language".

TODO: Apparently [5] §4 defines indexed W-types from W-types. They give a citation to a more formal definition and proof, but unfortunately, I can't find that paper on the internet (not even a citation!) There are also rumblings of extentionality or homotopy requiring alterations to the sense of the derivation. On the other hand, deriving W-types from indexed W-types is so straightforward it's hardly remarked on (just set the index set to $\mathbb{1}$). It's tempting to take indexed W-types as primary to the theory at that point.

10.5 TODO: unsorted inductive type stuff

TODO: mutual inductive families reduce to W-types; how about inductive-inductive types? what are inductive-recursive types?

11 Identity Types

Identity types (also called **equality types**[1]) are a key ingredient for formalizing mathematics, and thereby also for constructing those objects in dependently-typed programming languages.

$a =_A a'$	[1][7]
$I(A, a, a')$	[4]
$Id_A(a, a')$	gambino-hyland 2004

If an identity type is inhabited, it is at least inhabited by the **reflexive element**. In homotopy type theory, there may be additional distinct inhabitants, but we leave that for §11.3 (TODO: I think other type theories purposely trivialize the path space).

refl_x	[7]
r	[4]

An element of an identity type is also called an **equality proof** after the Curry-Howard Isomorphism.

I find that using the homotopy interpretation of identity types is less verbose.[1] An identity type is called a **path space**, and its elements are **paths**. This puts the focus squarely on *paths*, but to even name a path $p : x =_A y$, we must identify the **topological space** (or just **space**) A we are working in as well as two **endpoints** in that space $x, y : A$. This parameterization by space and endpoints is why identity types are manipulated as type families $\mathsf{Id}(\_,\_,\_) : (A : \mathcal{U}) \to A \to A \to \mathcal{U}$ with three indices.

For most types I program with, the corresponding topological space is what I might call a “dust”: a set of disconnected single points. When working with types like $\mathbb{Z}$ or $\mathbb{Q}$ represented as quotient sets, I visualize the corresponding space as a set of disconnected spheres (or balls): each distinct representation is a distinct point on a sphere, and if two representations are equivalent, then they are on the same sphere. I've yet to find a programing purpose for more complex topological spaces (e.g. disconnected donuts). These visualizations are a bit incorrect however; in the actual spaces corresponding to types, every identity path is a closed loop. Nevertheless, when considering one point (or both) as free to move, the metaphor works; it is when we consider paths with both endpoints fixed (as in paths between paths) that it breaks down, since two paths might take very different (i.e. not homotopic) routes (i.e. through a different sequence of holes TODO: winding number counts!?).

Now that we have a handle on how to form, introduce, and visualize identity types, how do we eliminate them, and what do we get out of it? The induction principle for identity types (called **path induction** in the homotopy interpretation) allows us to take a property that holds for refl and turn it into a property that holds for all paths. In other words, to prove a fact about all paths, it is sufficient to prove it for the null path at some unknown point (i.e. for all null paths); induction then allows us to drag around one end of the null path (stretching the path behind it) while maintaining that property. Compare in =-ELIM the types for f and $\mathsf{ind}_= f$, where we start with a function of a point that gives a property ξ of the null path $\xi\ a\ a\ \mathsf{refl}$, and end up with a function of a general path (and its endpoints) that gives the correspondingly general result $\xi\ x\ y\ p$. More tersely, “Thanks to path induction, that a property holds for refl is sufficient for it to also hold for all paths.”

$$\Gamma \vdash \frac{\xi : \prod_{x,y : \tau} (x =_\tau y) \to \mathcal{U} \qquad f : \prod_{a : \tau} \xi\ a\ a\ (\mathsf{refl}\ \tau\ a)}{\mathsf{ind}_= f : \prod_{\substack{x,y : \tau \\ p : x =_\tau y}} \xi\ x\ y\ p} \qquad (=\text{-ELIM})$$

A more ergonomic presentation would make the endpoint parameters (first two arguments) implicit. Path induction would then look more like other induction principles, where the result family is indexed by an element of the type we say we are inducting over. I just think it's too early now to start hiding details if the student wants to be confident in eliding arguments later: I want to see just how boring it is to specify these arguments for a little while *before* forgetting about them, and then it'll be easy to fill

[1]Compare “given an element of an identity type” or “given an equality proof” vs. “given a path”, and then just *try* not to think about any marginally more interesting statement.

them in when I want to check my work later. After all, the intuition here is not as obvious as for the notion of sets.

FIXME: move this, but just for completion, here's the principle with implict args:

$$\Gamma \vdash \frac{\xi : (x = y) \to \mathcal{U} \qquad f : \prod_{a\,:\,\tau} \xi_{\{a,a\}}\ \mathsf{refl}_{\{\tau,a\}}}{(\mathsf{ind}_{=\{\tau\}}\ f)_{\{x,y\}} : \prod_{p\,:\,x=y} \xi\ p}$$

Without identity types, the only notion of equality we had was judgmental equality, which is meta-theoretic. Once a type theory has identity types, we can now manipulate equality (specifically propositional equality) inside of the theory and prove facts about equality. The intriguing thing about identity types is that, given only (propositional) reflexivity, we can derive all the basic facts of (propositional) equality like symmetry (ex. 1) and transitivity (TODO: make a proof), as well as more advanced facts like function applicativity (TODO $(x = y) \to (f\ x = f\ y)$, [7] lemma 2.2.1) as opposed to generative (i.e. type theory is functional, not imperative), or the indiscernibility of identicals (TODO [7] lemma 2.3.1, which is the generalization for dependent functions). (TODO: I think these properties will be indispensible, but can I point at some examples?)

I think the thing that freaks me out about path induction is that it's so hard to imagine a reasonable inductive body that doesn't work. I mean, if your ξ actually uses its arguments to construct a path, how can passing x, x, refl fail? I suppose you can't do silly things such as: assuming z is some fixed point, I wouldn't (in general) be able to construct something of type $(\lambda x, y, p^{x=y}.\, x = z)\ x\ x\ \mathsf{refl}$. On the other hand, there are some silly things we can do, like *not* depend on the path in the result, e.g. $\xi \mapsto \lambda x, y, p.\, x + 1 : \prod_{x,y:\mathbb{N};\, p:x=y} \mathbb{N}$. I just don't know what that gives us that's useful.

11.1 TODO: Higher Inductive Types

TODO: higher inductive types (and quitient inductive types, higher inductive-inductive types, and so on). Kaposi's slides has a good list of these variants

TODO: is there a way to define graphs out of trees by identifying the results of following paths through the tree, and then also identifying elements by change of root?

11.2 TODO: Heterogeneous Equality

TODO: Idris uses this, but where is it in theory? The Idris docs gave an example of where it is needed, but IIRC it's not needed if one uses the full inductive principle rather than just the recursion principle.

11.3 TODO: Non-trivial Path Spaces

TODO: Until I can find some use for, say, $\mathbb{S}_1$ in programming, I think the types of homotopy theory are outside the scope of this work.

12 TODO: Constraint Types

TODO: typeclasses, row-polymorphism, algebraic effects

13 TODO: Syntactic Sugar

TODO nice let syntax

$$\mathsf{let}\,\overline{x_i^{\tau_i} = e_i}\,\mathsf{in}\,e' \triangleq (\lambda \overline{x_i^{\tau_i}}.\, e')\ \overline{e_i}$$

$$\begin{array}{l} \mathsf{let} \\ \quad f : \tau \to \sigma \\ \quad f\ x = e \\ \mathsf{in}\, e' \end{array}$$

TODO pattern matching

Part III
Case Studies

This part examines some well-recognized type theories. This was created as a place to experiment with my personal selections for notation, but it also serves as a "travel guide" for major ideas and presentations in the literature.

TODO: CIC as a foundation for Coq. In fact, the "cpdt book" (use google) has pointers for interesting properties.

14 Pure Type Systems

Definition 2. A **Pure Type System** (PTS) is a deductive system parameterized by a triple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ of **sorts**, **axioms** $\mathcal{A} \subseteq \mathcal{S}^2$, and **rules** $\mathcal{R} \subseteq \mathcal{S}^3$.

i) Allowing $\mathcal{X}$ to stand for some countably infinite set of variables, the syntax of a PTS is given by:

$$\begin{array}{rcl} x, y, z & \in & \mathcal{X} \\ s & \in & \mathcal{S} \\ e, f, \tau, \sigma, t & ::= & s \\ & \mid & x \\ & \mid & \lambda x{:}\tau.\, e \\ & \mid & f\ e \\ & \mid & (x : \tau) \to \tau' \end{array}$$

ii) Its reduction relation is the smallest relation containing

$$(\lambda x{:}\tau.\, e)\; t \longrightarrow [x \mapsto t]e$$

and (if an extensional PTS)

$$\lambda x{:}\tau.\, e^{()}\; x \longrightarrow e$$

The reflexive, transitive, symmetric, compatible closure of the reduction relation is the congruence relation

$$t \cong t'$$

iii) Then typing derivations are the smallest natural deduction system generated by

$$\frac{}{\Gamma \vdash s_1 : s_2}\;(s_1, s_2) \in \mathcal{A} \qquad \text{(Sort)}$$

$$\Gamma \vdash \frac{A : s}{x : A \vdash x : A}\; x \notin \mathrm{dom}(\Gamma) \qquad \text{(Var)}$$

$$\Gamma \vdash \frac{A : s \qquad t : \tau}{x : A \vdash t : \tau}\; x \notin \mathrm{dom}(\Gamma) \qquad \text{(Weaken)}$$

$$\Gamma \vdash \frac{x : \tau \vdash e : \sigma \qquad (x : \tau) \to \sigma : s}{\lambda x^{\tau}.\, e : (x : \tau) \to \sigma} \qquad \text{(Abs-Intro)}$$

$$\Gamma \vdash \frac{f : (x : \tau) \to \sigma \qquad t : \tau}{f\; t : \sigma} \qquad \text{(Abs-Elim)}$$

$$\Gamma \vdash \frac{\tau : s_1 \qquad x : \tau_1 \vdash \sigma : s_2}{(x : \tau) \to \sigma : s_3}\; (s_1, s_2, s_3) \in \mathcal{R} \qquad \text{(Prod)}$$

$$\Gamma \vdash \frac{e : \tau \qquad \tau' : s}{e : \tau'}\; \tau \cong \tau' \qquad \text{(Conv)}$$

I'm tempted to write Var and Weaken as a single rule $\Gamma \vdash x : \Gamma(x)$ However, this would allow ill-formed types to appear in the context. Perhaps this is ultimately not an issue, but I don't have a proof. The second premise of the conversion rule might also be dropped, assuming that congruence is sound.

14.1 PTSs Subsume the Lambda Cube

The systems of the Lambda Cube can be presented as pure type systems where the sorts are *types* and *kinds* ($\mathcal{S} = \{\star, \square\}$, resp.) and $\mathcal{A} = \{(\star : \square)\}$. Each of the systems has at least $(\star, \star, \star) \in \mathcal{R}$, and each feature of the cube adds an additional rule: *i*) polymorphism is $(\square, \star, \star)$, *ii*) type operators is $(\square, \square, \square)$, and *iii*) dependent types is $(\star, \square, \square)$.

A good presentation of this is given in [2]. In §3, Barendregt presents PTSs under the name **generalized type systems**; I have not seen this name used elsewhere. Be wary of Barendregt replacing s_3 by s_2 in the typing inference for dependent function spaces when discussing the lambda cube; it's good enough for the lambda cube, but not PTSs in general.

15 TODO: Calculus of Inductive Constructions

16 TODO: Martin-Löf Type Theory

17 TODO: Homotopy Type Theory

18 Proofs in the Theory

Example 1. As an example of using path induction, let's prove that whenever we have a proof of $x = y$, we can also prove $y = x$. In the theory, the proof is

$$\mathrm{sym}_{=} \overset{\text{def}}{=} \lambda A^{\mathcal{U}}.\, \mathsf{ind}_{=}\ (\lambda A^{\mathcal{U}}, x^{A}.\, \mathsf{refl}\ A\ x) : \underset{A:\mathcal{U};x,y:A}{\forall} (x =_{\!A} y) \to (y =_{\!A} x)$$

Since this term is little more than a use of $\mathsf{ind}_{=}$, we can informally abbreviate such a proof to "by induction". To check that our proof is correct, all we have to do is show that this term has the type we asserted by giving a typing derivation. The typing derivation is given in fig. 1. To reduce repetition in the derivation, we've made use of a couple helper definitions:

$$\mathrm{Sym}_{=} \overset{\text{def}}{=} \lambda A^{\mathcal{U}}, x^{A}, y^{A}, p^{x =_{\!A} y}.\, y =_{\!A} x$$
$$s \overset{\text{def}}{=} \lambda A^{\mathcal{U}}, x^{A}.\, \mathsf{refl}\ A\ x$$

which we instantiate for the =-ELIM rule's ξ, f respectively. Indeed, a slightly less terse informal proof can be "by induction using s at $\mathrm{Sym}_{=}$".

Part IV
Typesetting Tricks

- There can be too much space between an equals sign and its subscript, especially if the letterform has a forward slant at the front. Use `\!` to squeeze them back together.

	`x =_{\!\tau} y`	$x =_{\!\tau} y$
	`x =_{\!\!A} y`	$x =_{\!\!A} y$
	`x =_{\!B} y`	$x =_{\!B} y$
	`\fun{p^{x =_{\!\!A} y}}e`	$\lambda p^{x =_{\!\!A} y}.\, e$
c.f.	`\fun{p^{x =_A y}}e`	$\lambda p^{x =_A y}.\, e$

- A script-size colon can be squeezed by its neighbors. Use `\scrcolon` to force the space.

	`A:\U`	$A : \mathcal{U}$
	`\prod_{A\scrcolon\U}x=y`	$\prod_{A\,:\,\mathcal{U}} x = y$
c.f.	`\prod_{A:\U}x=y`	$\prod_{A:\mathcal{U}} x = y$

FIXME: This derivation has shown me where I can eliminate arguments from primitives like ind. I think all the rules need a thorough proof to see where I can abbreviate axioms. (At least abbreviate in theory; if type checking is going to be decidable, I might need more annotations.)

1	$A : \mathcal{U}$	
2	$x : A$	
3	$y : A$	
4	$p : x =_A y$	
5	$y =_A x$	=-form, 1, 2, 3
6	$\lambda x^A, y^A, p^{x =_A y}.\, y =_A x : \prod_{x,y:A} (x =_A y) \to \mathcal{U}$	Π-intro
7	$\mathrm{Sym}_=\ A : \prod_{x,y:A} (x =_A y) \to \mathcal{U}$	definition
8	$x : A$	
9	$\mathsf{refl}\ A\ x : x =_A x$	=-intro, 1, 8
10	$\mathsf{refl}\ A\ x : (\lambda x^A, y^A, p^{x =_A y}.\, y =_A x)\ x\ x\ (\mathsf{refl}\ A\ x)$	multiple β^{-1}
11	$\mathsf{refl}\ A\ x : (\mathrm{Sym}_=\ A)\ x\ x\ (\mathsf{refl}\ A\ x)$	definition and β^{-1}
12	$\lambda x^A.\, \mathsf{refl}\ A\ x : \prod_{x:A} \mathrm{Sym}_=\ A\ x\ x\ (\mathsf{refl}\ A\ x)$	Π-intro
13	$s\ A : \prod_{x:A} \mathrm{Sym}_=\ A\ x\ x\ (\mathsf{refl}\ A\ x)$	definition and β^{-1}
14	$\mathsf{ind}_=\ (s\ A) : \prod_{\substack{x,y:A \\ p:x =_A y}} \mathrm{Sym}_=\ A\ x\ y\ p$	=-elim, 7, 13
15	$\lambda A^{\mathcal{U}}.\, \mathsf{ind}_=\ (s\ A) : \prod_{\substack{A:\mathcal{U};\, x,y:A \\ p:x =_A y}} \mathrm{Sym}_=\ A\ x\ y\ p$	Π-intro
16	$\mathrm{sym}_= : \forall_{A:\mathcal{U};x,y:A} (x =_A y) \to (y =_A x)$	definition and β

Figure 1: Typing derivation for Path Symmetry

- Commas in script size can end up with wonky spacing; FIXME: I don't know what to do about it realistically.

`\Pitype{x,y}{A} B`	$\prod_{x,y:A} B$
`\Pitype{x\mkern -1.4mu ,\mkern 1.4mu y}{A} B`	$\prod_{x\mkern -1.4mu ,\mkern 1.4mu y:A} B$

- Subscripts in display mode can get very long. Use `\mathclap` to collapse the space before and after, but additional spacing might be needed to avoid neighbors.

`\prod_{x:\tau,y:\sigma}x=y`	$\displaystyle\prod_{x:\tau,y:\sigma}x=y$
`\prod_{\mathclap{x:\tau,y:\sigma}}x=y`	$\displaystyle\prod_{\mathclap{x:\tau,y:\sigma}}x=y$
`\prod_{\tau:\U}\;\; \prod_{\mathclap{x:\tau,y:\sigma}}x=y`	$\displaystyle\prod_{\tau:\mathcal{U}}\;\;\prod_{\mathclap{x:\tau,y:\sigma}}x=y$

- Subscripts underneath can get jammed against the bottom of a big operator. `\substack` alleviates this.

`\prod_{a \in s}`	$\displaystyle\prod_{a \in s}$
`\prod_{\substack{a \in s}}`	$\displaystyle\prod_{\substack{a \in s}}$

Part V
TODO: Unsorted

Higher-order function: a function which takes one or more functions as arguments. Higher-order types: allows for the definition of (non-nullary) type operators a.k.a. type constructors. Higher-kinded: type variables are allowed to range over type operators as well as simple types. Higher-rank: quantifiers are allowed to appear to the left of function arrows.

I tend to include η-conversion in my calculi; it just makes sense that deconstructing a constructor (eliminating an introduction) does nothing.

In axioms for typing judgments, threading the context around is often boring and obscures the substance. Instead, let's put the shared part on the right when possible:

$$\Gamma \vdash \frac{f : A \to B \qquad a : A}{f\,a : B} \tag{\textsc{Abs-Intro}}$$

and extend it in each premise/conclusion as needed:

$$\Gamma \vdash \frac{x : A \vdash e : B}{\lambda x.\, e : B} \tag{\textsc{Abs-Elim}}$$

Though to be honest, we might not even need to write the context most of the time, at least once we get used to the notation. There are some rules that do need it, though:

$$\Gamma \vdash \frac{e : A \qquad T : \tau}{x : T \vdash e : A} \text{ when } x \notin \operatorname{dom}(\Gamma) \tag{\textsc{Weaken}}$$

And I haven't examined if there'd be any confusion between omitting a context vs. specifically having no context.

Contexts have some interesting notations. How do you write empty context? $\epsilon, \varepsilon, \diamond, \varnothing, \emptyset$, or with no ink at all? How to you append to contexts? A comma, $\cup$, $+$? How do you catenate contexts? A comma, mere adjacency, $\cup$? Contexts in general are ordered finite maps, but outside of dependent types, order is often irrelevant. How do you check the domain of a context? $x \in \Gamma$ or $x \in \mathrm{dom}(\Gamma)$? How to you lookup a binding from a context? $x : A \in \Gamma$ or $\Gamma(x) = A$? When $\Gamma(x) = A$, can you write $\Gamma(x)$ as a synonym for A?

TODO: define abstract syntaxes by a BNF-like notation

TODO System U, UU^{-}, and Girard's paradox with proof http://www.cs.cmu.edu/~kw/scans/hurkens95tlca.pdf

TODO: typing judgments; most everyone uses $a : A$, but I've seen $a \in A$[1]

TODO: normal forms can also be called **canonical elements**[4]

TODO: Telescopes from §2.5 of Dyber Inductive Sets and Families

TODO: It suddenly seems to me that the Y-combinator just smashes stuff with a hammer to get recursion. If you know what sort of thing you'll be recursing (or performing induction) over, the the various ind_σ principles from dependent type theory illustrate so much more delicate structure hiding behind Y.

TODO: I want to come back to ornaments at some point, as they seem pretty cool

TODO: missing arguments $f(\_, x)$, $f(-, x))$, $f(\cdot, x)$

TODO: Is there anything for, say, defining binary operations on an abstract type? Like, let's take two implementations of $\mathbb{Z}$: one representing ints as $a - b$; the other as zero, the decrement of a non-positive, or the increment of a non-negative. Then, we define the integers proper as an existential type, and show we can pack the two integer representations appropriately. What eliminators should the existential expose? How can we add more eliminators after the fact? We'll want multiple eliminators because some operations may be more efficient with one eliminator than another (i.e. addition is faster on $a - b$ representation, but determining sign is faster on an inc/dec representation). Is there a way to decide whether to press on with an inefficient representation rather than just use an isomorphism? I've stated this for integers, but complex numbers are a classic case as well.

TODO: Lean allows parameter types to be listed as if they were arguments, rather than sloughing all the parameter types to the lhs of some arrows.

```
comp {a b c : Type} (b -> c) (a -> b) :: a -> c
comp f g x = g (f x)
```

This is very handy, and kinda related to transforming between $\mathsf{ind} f x : \tau$ conclusions and $\mathsf{ind} f : (x : \sigma) \to \tau$ conclusions.

TODO motive and methods for induction principles: what are they?

TODO: Strong normalization is handy because we don't have to worry about evaluation strategy: "given an arbitrary term, any reduction sequence ends in a normal form". Weak normalization still rules out non-terminating programs, but not necessarily non-terminating evaluators: "given an arbitrary term, there is at least one reduction sequence that ends in a normal form". In between there should be some additional notions: "given an arbitrary term, there is a reduction sequence that results in a normal form which is accessible given a local reduction strategy". By reduction strategy I mean that the syntactic form of a term uniquely identifies which rewrite rule is used, and by local I mean that there are a finite number of syntactic patterns (possibly involving deterministic side-conditions) that are searched for. I'm pretty sure "reduction strategy" is actually something related in the literature.

TODO: notation for named holes, since they're useful in proof assistants

TODO: To understand dependent functions, I always pick this example, which uses only high-school math.

i) Consider the functions which take a real number as input and produces a real number as output. These are the ordinary functions like $f(x) = x - 3, f(x) = x^2, f(x) = e^x$, we work with in elementary algebra. If f is such a function, we might use the notation

$$f : \mathbb{R} \to \mathbb{R}$$

to express this idea more quickly. This notation used in both standard mathematics as well as type theory.

ii) But consider the functions which, as before, take a real input and produce a real output, but now we additionally constrain the output so that it is always less than or equal to whatever the input was. Intuitively, the functions are those that, when plotted, never go above the line $y = x$. TODO: draw a plot using pgfplot. We can see that most of the examples we gave before do not fall into *this* class of function. One function we saw before ($f(x) = x - 3$) follows the rules, as do functions like $f(x) = x$, $f(x) = max(x, 0)$ and so on. However, the other two example functions from (i) *do not* follow the rules, and so are not included in this class of function: it's easy to see that $f(x) = e^x$ is always above the line, and $f(x) = x^2$ goes above the line as soon as $x > 1$.

Can we work with this class of functions more quickly than with prose? Standard mathematics can identify these functions as a set $\{f \in \mathbb{R} \to \mathbb{R} \mid \forall x. f(x) \leq x\}$, but this approach has its downsides. For one, I personally think set theory is something of a verbose foundation for mathematics which encourages more use of informal prose in definitions, theorems, and proofs. The more pressing issue is that there's no $f : \ldots$ notation corresponding to that used in (i) which would make smooth the transition from simple classes of functions to constrained classes.

iii) In type theory, proofs are first-class objects and can be applied and manipulated formally. If we need to know that the output is always less than the input, the method that first comes to my mind is to attach the required proof to the output. That is, we want something of the form

$$f : \mathbb{R} \to \mathbb{R} \times (\boxed{\text{OUTPUT}} \leq \boxed{\text{INPUT}}),$$

but what should we fill in as $\boxed{\text{INPUT}}$ and $\boxed{\text{OUTPUT}}$? They obviously need to refer to the input and the output reals, so we somehow need to give names to these things. Dependent functions allow us to give a name to the input, and makes that variable available to form the output type. Let's do that and fill in the $\boxed{\text{INPUT}}$ hole:

$$f : (x : \mathbb{R}) \to \mathbb{R} \times (\boxed{\text{OUTPUT}} \le x).$$

The output real—as opposed to the output proof—is just the first part of the pair, but dependent pairs allow us to name this half of the pair and use that variable in the second half. This is exactly what we need to fill in $\boxed{\text{OUTPUT}}$:

$$f : (x : \mathbb{R}) \to (y : \mathbb{R}) \times (y \le x).$$

Putting it all together, we read the above as "f is a function which takes a real input (x) to an ordered pair consisting of the real "output" (y) along with a proof that the output is less than or equal to the input ($y \le x$)". This is very close to the prose as we wrote in (*ii*), so it seems like this is a good translation of the idea.

iv) There is another, perhaps more accurate way of formalizing our desired class of functions from (*ii*). The idea is to provide first the function, then a proof about that function. If you'll allow me the liberty of extending the $a : \mathbb{Z}$ notation (which names a value from a set) to also be able to name a proof of a proposition, then standard mathematics might write

$$\begin{gathered} f : \mathbb{R} \to \mathbb{R}, \text{ and} \\ lte : \forall x.\, f(x) \le x, \end{gathered}$$

which in type theory would be written as the specification of a single dependent pair:

$$\langle f, lte \rangle : (f : \mathbb{R} \to \mathbb{R}) \times (\forall x.\, f(x) \le x).$$

Here we have an ordinary function on reals f, but packaged with a proof that for any input ($\forall x$), its outputs ($f(x)$) are less than its inputs ($f(x) \le x$). In fact, although we use the logical notation $\forall$, in type theory universal quantification is encoded with a dependent function. In this case: $\forall x.\, f(x) \le x \rightsquigarrow (x : \boxed{\text{T}}) \to f(x) \le x$. In this case, we know x must be a real number, since $f(x)$ must be well-typed, so we can fill in $\boxed{\text{T}}$ with $\mathbb{R}$ to obtain this fully type-theoretic specification:

$$\langle f, lte \rangle : (f : \mathbb{R} \to \mathbb{R}) \times ((x : \mathbb{R}) \to f(x) \le x).$$

This makes it clear that what we have is a pair of functions, one of which produces the values we care about, and the other provides the proofs we need for each value. It's easy to see by translating this type back to prose, that this is *also* a good translation of the original idea.

v) Indeed, the types demonstrated in (*iii*) and (*iv*) are *equally good* translations of the idea in (*ii*), and it even turns out that this equivalence can be formalized. If we read functions as exponentials (i.e. $A \to B$ is like B^A), then their equivalence is just an instance of familiar algebraic law $(y \times z)^x = y^x \times z^x$, taking x as the input type, y as the output type, and z as the proof type. This metaphor can be expressed formally in category theory.

vi) Note that so far, I've only demonstrated *total* functions. Although standard mathematical practice would allow $f(x) = \ln(x)$ to be part of the class of functions from (*ii*), but the type-theory translations would not be accurate because $\ln(x)$ is a *partial* function. In standard mathematical practice arrow often denotes partial functions, but in type theory it is *always* total. In type theory, we would say

$$f : \mathbb{R} \to \mathbb{1} + \mathbb{R}, \text{ and}$$
$$lte : (x : \mathbb{R}) \to \mathsf{case}\, f(x)\, \mathsf{of}\, \{\iota_1(0_{\mathbb{1}}) \Rightarrow \mathbb{1}; \iota_2(y) \Rightarrow y \leq x\}$$

to express that if $f(x) = y$ is defined we need $y \leq x$, but if it's undefined we don't need to do anything special[2]. I'm sure there's a clever way to write this more concisely using point-free programming, but I won't do it just yet.

vii) Even this simple example can be the basis for important applications. A similar type of function $(f : \mathbb{N} \to \mathbb{N}) \times (\forall n. f(n) < n)$ would allow us to make proofs by descent. Likewise, only (relatively) small changes are needed to express the class of linear functions in complexity theory

$$(f : \mathbb{R}^{0+} \to \mathbb{R}^{0+}) \times (x_0 : \mathbb{R}^{0+}) \times (\varepsilon : \mathbb{R}^{+}) \times (x \geq x_0 \to \varepsilon \cdot f(x) \leq x),$$

writing $\mathbb{R}^{0+}$ for non-negative reals $\mathbb{R}^{+}$ for positive reals. For those not familiar with type theory, what I've done is added a "base point" x_0 and "multiplicative factor" ε to the dependent tuple, conditioned the proof so that it need only be demonstrated for inputs above the base point $(x \geq x_0 \to \ldots)$, and relaxed the proof so that the output can be scaled by the multiplier $\varepsilon \cdot f(x) \leq x$. I don't know about you, but it took me not insignificant effort in school to formally understand the moving pieces of big-O notation; with this type in front of me, I can clearly see where each part is introduced and over what scope it stays constant, and why. Presumably, the notorious ε–δ definition of limits would also be clearer in type theory than informally; such an exercise is left to the reader.

TODO: it seems like dependent pairing should be associative, but I doubt the notation suports it. Consider

$$a : A \times (b : B(a) \times C(a, b)) \quad \text{versus}$$
$$(a : A \times b : B(a)) \times C)$$

What if I could develop rules to make such notation sensible?

TODO: Is there a way to implicitly carry if-then-else evidence through to the branches?

[2]we can just provide the unit value $0_{\mathbb{1}}$ of the unit type $\mathbb{1}$

```
arrIndex :: (arr : Array n a) -> (n : USize) -> {inBounds :: n < arr.length} -> IO a
arrIndex = ...
main = do
  xs <- arrayFromList [1,2,3,4,5]
  n <- readUSize -- i.e. from the command line
  if (n < xs.length)
    then print (arrIndex xs n) -- the term ‘n < xs.length‘ should not only give a ‘Bool‘
    else putStrLn "Out of bounds" -- the proof term from the then-branch os not availabl
```

I already was thinking in terms of `class Truthy t where toPred :: a -> Bool`, but what about a proofy typeclass

```
class Proofy t where
  type Proves t :: Prop
  toBool :: t -> Bool
  toProof :: (x :: t) -> (toBool x = True) -> Proves t
```

where I would want the proof to always be erased before runtime.

References

[1] Roland Backhouse et al. "Do-it-yourself type theory". In: *Formal Aspects of Computing* 1 (Mar. 1989), pp. 19–84. DOI: 10.1007/BF01887198.

[2] Henk Barendregt. "Introduction to generalized type systems". In: *Journal of Functional Programming* 1.2 (1991), pp. 125–154. DOI: 10.1017/S0956796800020025.

[3] Ambrus Kaposi and András Kovács. "Signatures and Induction Principles for Higher Inductive-Inductive Types". In: *Log. Methods Comput. Sci.* 16.1 (2020). DOI: 10.23638/LMCS-16(1:10)2020. URL: https://doi.org/10.23638/LMCS-16(1:10)2020.

[4] Per Martin-Löf. *Intuitionistic type theory. Notes by Giovanni Sambin.* Vol. 1. Studies in Proof Theory. Bibliopolis, 1984, pp. iv+91. ISBN: 88-7088-105-9.

[5] Kent Petersson and Dan Synek. "A Set Constructor for Inductive Sets in Martin-Löf's Type Theory". In: *Category Theory and Computer Science.* Berlin, Heidelberg: Springer-Verlag, 1989, pp. 128–140. ISBN: 354051662X.

[6] Benjamin C. Pierce. *Types and Programming Languages.* 1st. The MIT Press, 2002. ISBN: 0262162091.

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