Dependent Type Theory for Programming and Proving*

Robert Harper

Spring 2024

1 Introduction

Simple, or *non-dependent*, type theories feature a complete separation between types and programs. Types, given *a priori*, are used to classify programs based on their execution behavior, which is also specified in advance. More precisely, types determine when two programs behave the same way, and hence are considered equal. In richer languages types themselves admit non-trivial notions of equality, always adhering to the general principle that equal types classify the same behaviors. Such languages are said to be *phase separated* in that type checking is regarded as a *static*, or *compile-time*, or *verification-time*, notion, whereas execution is regarded as a *dynamic*, or *run-time*, notion. The vast majority of extant programming languages are phase-separated in this sense.

By contrast *dependent* type theories exhibit no such separation of types from programs. In fact types are themselves programs that have the special status of being used as classifiers of other programs—and they themselves are also so-classified. Dependent type theories are *phase integrated* in that there is no clear separation between types and programs, rather types are simply a certain class of programs that admit interpretation as classifiers. Such type theories are distinctly expressive, so much so that they serve not only to classify data objects, but are also precise enough to serve as specifications of program behavior. Thus, a type such as "sequence of length n" classifies finite sequences of values whose length is $n \geq 0$, and arrow types such as those between sequences of length n and sequences of length $2 \times n$ may be regarded as specifications of program behavior. Transposing the emphasis, programs are regarded as evidence that a type *qua* proposition is *true* in the Brouwerian sense of being realized by a program.

One role of dependent types in programming languages is as a rich language for program modules, separable components that may be selectively combined to form larger components, ultimately complete programs. Such languages, though reliant on dependent types for their expressive power, exhibit a *phase distinction* between the static and dynamic aspects of a program module, the static aspects being types and the dynamics aspects being executable programs. The distinction lies in the treatment of equality of components: whereas two modules may differ when considered in full, they may coincide when their run-time aspects are disregarded, thereby isolating only their compile-time aspects. Such a distinction is crucial for supporting well-established program development methodologies, particularly those based directly on combining *statically coherent* components to form other components. Given such a phase distinction, violations of such coherence constraints are visible at compile time, ensuring that violations are caught when the program is composed, rather than once delivered to a user.

The development of these ideas will proceed in stages, first considering only value types, then considering computation types, and finally the phase distinction. The core concept of dependent type theory is that of a *family of types* indexed by the elements of another type or types. A non-indexed type is a degenerate family with no indexing; otherwise, a family is, informally, a mapping from a type into the "multiverse" of all types.[1] For technical reasons families are usually defined in an iterated ("curried") form, so as to avoid the reliance on product types in their definition, though once those are available, a family can always be presented in simultaneous ("uncurried") form. Families are closed under generalizations of the function and product types considered in the non-dependent setting, and are enriched to include inductive and coinductive types, albeit with dependent forms of their elimination and introductory forms, respectively. A crucial question is the status of equality as a family of types; there are several extant notions with different properties. Finally, a stratified hierarchy of *universes*, types whose elements are types, is considered, affording considerable expressive power and providing convenient technical devices for consolidating familes of types with families of elements. Then type theory is equipped, in the manner of cbpv, with effectful computation types, which are distinguished to avoid compromising the very notion of a family, or the fundamental propositions-as-types principle. The static/dynamic phase distinction collapses the elements of a value type, and of all computation types, so as to isolate the compile-time significance of a program module from its run-time aspects. The associated notion of an extension type provides support for the notion of a sharing specification governing the coherent composition of programs from components.

2 Pure Dependent Type Theory

A formalism for dependent type theory defines the following judgment forms:

1. $\Gamma \vdash A\ \mathsf{type}$, stating that A is a type in context Γ;
2. $\Gamma \vdash A \equiv A'\ \mathsf{type}$, stating that A and A' are equal types in context Γ;
3. $\Gamma \vdash M : A$, stating that M is a term of type A;
4. $\Gamma \vdash M \equiv M' : A$, stating that M and M' are equal elements of type A.

Whereas the latter two judgments are familiar from non-dependent type theory, the first two are manifestations of the inter-relatedness between types and their elements that is characteristic of dependent type theory.

Any set of rules defining these judgments should be formulated to ensure that the following *well-formation* conditions are true:

1. If $\Gamma \vdash A \equiv A'\ \mathsf{type}$, then $\Gamma \vdash A\ \mathsf{type}$ and $\Gamma \vdash A'\ \mathsf{type}$.
2. If $\Gamma \vdash M : A$, then $\Gamma \vdash A\ \mathsf{type}$.
3. If $\Gamma \vdash M \equiv M' : A$, then $\Gamma \vdash M : A$ and $\Gamma \vdash M' : A$.

Contrarily, the well-formation of the context is *pre-supposed* in these situations! That is, the successive types of the variables in a context are assumed to be well-formed types in the prefix of the context up to, but not including, that variable.

[1] The motivation for the peculiar terminology will emerge as part of the development.

$$\textsc{type-refl}\ \frac{\Gamma \vdash A\ \mathsf{type}}{\Gamma \vdash A \equiv A\ \mathsf{type}} \qquad \textsc{type-sym}\ \frac{\Gamma \vdash A \equiv A'\ \mathsf{type}}{\Gamma \vdash A' \equiv A\ \mathsf{type}} \qquad \textsc{type-trans}\ \frac{\Gamma \vdash A \equiv A'\ \mathsf{type} \qquad \Gamma \vdash A' \equiv A''\ \mathsf{type}}{\Gamma \vdash A \equiv A''\ \mathsf{type}}$$

$$\textsc{elt-refl}\ \frac{\Gamma \vdash M : A}{\Gamma \vdash M \equiv M : A} \qquad \textsc{elt-sym}\ \frac{\Gamma \vdash M \equiv M' : A}{\Gamma \vdash M' \equiv M : A} \qquad \textsc{elt-trans}\ \frac{\Gamma \vdash M \equiv M' : A \qquad \Gamma \vdash M' \equiv M'' : A}{\Gamma \vdash M \equiv M'' : A}$$

$$\textsc{elt-resp}\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash A \equiv A'\ \mathsf{type}}{\Gamma \vdash M : A'} \qquad \textsc{eqv-resp}\ \frac{\Gamma \vdash M \equiv M' : A \qquad \Gamma \vdash A \equiv A'\ \mathsf{type}}{\Gamma \vdash M \equiv M' : A'}$$

$$\textsc{var-of}\ \frac{}{\Gamma\, x : A\, \Gamma' \vdash x : A} \qquad \textsc{var-eqv}\ \frac{\Gamma \vdash x : A}{\Gamma \vdash x \equiv x : A}$$

Figure 1: Structural Rules of Dependent Type Theory

The *structural rules* for a dependent type theory are given in Figure 1. These rules are to be included in any dependent type theory to ensure that the following general principles are upheld:

1. Judgmental equality of types and their elements is an *equivalence relation*.
2. Formation and equality of elements *respects equality* of their classifiers: equal types determine equal elements and their equality.
3. Variables inhabit the types according to their declaration in the context. Consequently, the induced entailment between types is *reflexive*.
4. *Substitution* of elements of a type for variables of that type must be valid for all forms of judgment. Consequently, the induced entailment between types is *transitive*.
5. The dependence of types and terms on variables must be *functional* in that substitution of equal elements must yield equal results.

Reflexivity and substitution suffice to ensure that *weakening* (adding additional variables) and *contraction* (consolidating two variables of the same type) are admissible.[2] The functionality condition for types is fundamental to the very idea of dependency: the occurrence of variables within types is that of a function from a type to the multiverse of types.

There are many variations on the formulation of dependent type theory according to various criteria such as convenience for usage in an implementation, and according to the formulation of concepts such

[2]The proof of both properties follows from a generalization of substitution to simultaneous instantiation of the variables in a context by terms in another context. The rules must be formulated with this in mind; for example, the "extra" context beyond the declaration of the variable in the variable rule is there to "build in" weakening to ensure that it be admissible.

as inductive types. These variations are far too numerous to consider here; the present development is concerned only with the skeletal apparatus of dependent type theory. The formulation considered here relies on the admissibility of, say, substitution and weakening, which means that they hold for a given set of rules, and must be reconsidered whenever the theory is extended with new constructs.

Although all of the value types in a non-dependent setting can be carried over to the dependent setting without change, to get their full expressive power their dependent counterparts usually have much richer—and more flexible—typing rules. A good example is the type bool of booleans, whose introductory forms are true and false, and whose elimination is the conditional branch, $\mathsf{if}(e;e_1;e_2)$. In a non-dependent setting both branches of the conditional must be of the same type, because it is not possible to predict the outcome of the branch statically. However, in a dependent setting we have the possibility—because expressions can appear in types—to give a much more powerful typing rule, and associated equations, as given in Figure 2. The important point is that the type of the conditional is dependent on the condition on which it is branching, which raises the possibility that the result type can also branch on the same condition to determine the type for each branch. This is achieved using a type-level version of the conditional, called a *large elimination form*, that yields a type instead of a term on each branch.[3] Thus, one may judge that $\mathsf{if}(M;7;\mathsf{true})$ is of type $\mathsf{If}(M;\mathsf{nat};\mathsf{bool})$! It is important to study the rules in Figure 2 very carefully to see how this is achieved. In particular note well that the generic rule of type equivalence plays an essential role. In general the elimination form of any inductive type—including the empty type, any sum type, and the type of natural numbers—is similarly general, and relies—for the moment—on the "large" elimination forms to define a family that takes full advantage of the dependency of the type of the elimination on the expression being eliminated.

It is worth noting that, even for the type of booleans, the equations given in Figure 2 are incomplete in that there are true equtaions that cannot be proved using these rules. An example of such an equation is $\mathsf{if}(M;N;N) \equiv N : A$, stating that if both branches of a conditional are the same, then the there is no need to branch at all. It is possible, though technically rather involved, to prove that this equation is not derivable from the given rules, and is therefore "missing" from the theory. In the special case of booleans, or any other finite type, it is possible to rectify this shortcoming by introducing the following principle of *boolean induction* for proving equations involving a boolean variable:

BOOL-IND
$$\frac{\Gamma \vdash M : \mathsf{bool} \qquad \Gamma, x : \mathsf{bool} \vdash N : B \qquad \Gamma, x : \mathsf{bool} \vdash N' : B \qquad \Gamma \vdash [\mathsf{true}/x]N \equiv [\mathsf{true}/x]N' : [\mathsf{true}/x]B \qquad \Gamma \vdash [\mathsf{false}/x]N \equiv [\mathsf{false}/x]N' : [\mathsf{false}/x]B}{\Gamma \vdash [M/x]N \equiv [M/x]N' : [M/x]B}$$

That is, if two terms, N and N', with a free variable of boolean type coincide on true and an false, then they are equal for any boolean. Using this rule equations such as the one mentioned are derivable. So why not add it to the theory? There are two reasons, one practical, the other conceptual. As a practical matter the principle of bool-induction involves a bifurcation of proof obligations, one for true, one for false. When nested inside one another, each such bifurcation introduces another factor of two, and quickly becomes unmanageable. For this reason, bool-induction is not normally included in the equations of a type theory. As a conceptual matter, the idea does not scale to other inductive types, the premier example being the natural numbers, for the simple reason that the inductive hypothesis in equality proof would require an equational *assumption*, and these are not available in type theory as presented here. This question will be reconsidered once "identity types" are considered; for now, let us

[3] In due course the "large" conditional will be reduced to an ordinary conditional whose branches are elements of a "universe" of types.

$$\textsc{true}\ \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \qquad \textsc{false}\ \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}$$

$$\textsc{if}\ \frac{\Gamma \vdash M : \mathsf{bool} \qquad \Gamma, x : \mathsf{bool} \vdash B\ \mathsf{type} \qquad \Gamma \vdash M_1 : [\mathsf{true}/x]B \qquad \Gamma \vdash M_2 : [\mathsf{false}/x]B}{\Gamma \vdash \mathsf{if}(M;M_1;M_2) : [M/x]B}$$

$$\textsc{if-true}\ \frac{\Gamma, x : \mathsf{bool} \vdash B\ \mathsf{type} \qquad \Gamma \vdash M_1 : [\mathsf{true}/x]B \qquad \Gamma \vdash M_2 : [\mathsf{true}/x]B}{\Gamma \vdash \mathsf{if}(\mathsf{true};M_1;M_2) \equiv M_1 : [\mathsf{true}/x]B}$$

$$\textsc{if-true}\ \frac{\Gamma, x : \mathsf{bool} \vdash B\ \mathsf{type} \qquad \Gamma \vdash M_1 : [\mathsf{true}/x]B \qquad \Gamma \vdash M_2 : [\mathsf{true}/x]B}{\Gamma \vdash \mathsf{if}(\mathsf{true};M_1;M_2) \equiv M_2 : [\mathsf{true}/x]B}$$

$$\textsc{If}\ \frac{\Gamma \vdash M : \mathsf{bool} \qquad \Gamma \vdash A_1\ \mathsf{type} \qquad \Gamma \vdash A_2\ \mathsf{type}}{\Gamma \vdash \mathsf{If}(M;A_1;A_2)\ \mathsf{type}} \qquad \textsc{If-true}\ \frac{\Gamma \vdash A_1\ \mathsf{type} \qquad \Gamma \vdash A_2\ \mathsf{type}}{\Gamma \vdash \mathsf{If}(\mathsf{true};A_1;A_2) \equiv A_1\ \mathsf{type}}$$

$$\textsc{If-false}\ \frac{\Gamma \vdash A_1\ \mathsf{type} \qquad \Gamma \vdash A_2\ \mathsf{type}}{\Gamma \vdash \mathsf{If}(\mathsf{false};A_1;A_2) \equiv A_2\ \mathsf{type}}$$

Figure 2: Booleans in Dependent Type Theory (Selected Rules)

simply say that it is not practical to admit induction as a means of proving equations involving variables of an inductive type.

Similarly, the function type is generalized to permit the range type of an application to be depend on the argument value. Thus, the type $A \to B$ becomes the dependent form $x : A \to B$, where B may depend on x of type A. Correspondingly, the type of an application substitutes the argument itself into the result type of the function. Similarly, the product type $A \times B$ becomes the dependent form $x : A \times B$, again with B depending on x of type A. Correspondingly, the type of the second projection substitutes the first projection itself into the type of the second component. Rules for these generalizations are given in Figures 3 and 4. The dependent function type degenerates into the ordinary function type in the case that the range type is independent of the argument, and similarly the dependent product type degenerates into the ordinary product type in the case that the type of the second component does not depend on the first component. It is possible to consider η-like principles for both the product and function types; these are left as an exercise. Transposed into logic the dependent function type corresponds to universal quantification, a view that is especially natural when the domain type is a "data" type as opposed to a proposition. Similarly, the dependent product type corresponds to existential quantification in the full-throated constructive sense in which the witness to the existential is provided along with the proof that it satisfies the property given as second component. Constructive negation is

$$\text{PI}\quad \frac{\Gamma \vdash A_1\ \mathsf{type} \qquad \Gamma, x : A_1 \vdash A_2\ \mathsf{type}}{\Gamma \vdash x : A_1 \to A_2\ \mathsf{type}} \qquad \text{LAM}\quad \frac{\Gamma \vdash A_1\ \mathsf{type} \qquad \Gamma, x : A_1 \vdash M_2 : A_2}{\Gamma \vdash \lambda(x.M_2) : x : A_1 \to A_2}$$

$$\text{APP}\quad \frac{\Gamma \vdash M : x : A_1 \to A_2 \qquad \Gamma \vdash M_1 : A_1}{\Gamma \vdash \mathsf{ap}(M;M_1) : [M_1/x]A_2} \qquad \text{BETA}\quad \frac{\Gamma \vdash M_1 : A_1 \qquad \Gamma, x : A_1 \vdash M_2 : A_2}{\Gamma \vdash \mathsf{ap}(\lambda(x.M_2);M_1) \equiv [M_1/x]M_2 : [M_1/x]A_2}$$

Figure 3: Dependent Function Types (Selected Rules)

$$\text{SIGMA}\quad \frac{\Gamma \vdash A_1\ \mathsf{type} \qquad \Gamma, x : A_1 \vdash A_2\ \mathsf{type}}{\Gamma \vdash x : A_1 \times A_2\ \mathsf{type}} \qquad \text{PAIR}\quad \frac{\Gamma \vdash M_1 : A_1 \qquad \Gamma \vdash M_2 : [M_1/x]A_2}{\Gamma \vdash \langle M_1, M_2\rangle : x : A_1 \times A_2}$$

$$\text{FST}\quad \frac{\Gamma \vdash M : x : A_1 \times A_2}{\Gamma \vdash M \cdot 1 : A_1} \qquad \text{SND}\quad \frac{\Gamma \vdash M : x : A_1 \times A_2}{\Gamma \vdash M \cdot 2 : [M \cdot 1/x]A_2}$$

$$\text{FST-PAIR}\quad \frac{\Gamma \vdash M_1 : A_1 \qquad \Gamma \vdash M_2 : [M_1/x]A_2}{\Gamma \vdash \langle M_1, M_2\rangle \cdot 1 \equiv M_1 : A_1} \qquad \text{SND-PAIR}\quad \frac{\Gamma \vdash M_1 : A_1 \qquad \Gamma \vdash M_2 : [M_1/x]A_2}{\Gamma \vdash \langle M_1, M_2\rangle \cdot 2 \equiv M_2 : [M_1/x]A_2}$$

Figure 4: Dependent Product Types (Selected Rules)

definable as $\neg A \stackrel{\text{def}}{=} A \to \mathsf{void}$, a transformation of a hypothetical proof of A into a proof of falsehood. Famously, $\neg\neg A$ is weaker than A itself, though in the presence of continuation effects the two can be reconciled.

Exercise 1. *Formulate η-like equations for dependent function and product types.*

So far there are no examples of dependent types that would make use of the infrastructure established so far. As with conventional textbook logic, the *identity* (or *equality*) type provides a fundamental example.[4] The identity type, $\mathsf{Id}[A](M;N)$, is axiomatized as the least reflexive binary relation on elements of type A, which is presented in Figure 5. Whereas the introductory form is straightforward, the eliminatory form is infamously obscure. However, all it says is to prove a property B of a given proof P of the identity of M and N, it suffices to prove that B is true for all possible instances of reflexivity, with the proof given $x.Q$. The stated equation specifies that if the proof P is, in fact, an instance of reflexivity, then the induction form is definitionally equivalent to the corresponding instance of Q.

Though beguiling, the difficult with this formulation of equality is that it reduces equality to *syntactic identity*. In particular, when A is a function type $A_1 \to A_2$, it deems two functions F and G of this type to be equal exactly when they are *identical*, up to judgmental equality. But according to this

[4]Which terminology is used depends on the formulation, of which two main examples are extant.

$$\text{ID}\quad \frac{\Gamma \vdash A\,\mathsf{type} \qquad \Gamma \vdash M : A \qquad \Gamma \vdash N : A}{\Gamma \vdash \mathsf{Id}[A](M;N)\,\mathsf{type}} \qquad \text{REFL}\quad \frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{refl}[A](M) : \mathsf{Id}[A](M;M)}$$

$$\text{IND}\quad \frac{\begin{array}{c}\Gamma \vdash M : A \qquad \Gamma \vdash N : A \qquad \Gamma \vdash P : \mathsf{Id}[A](M;N) \\ \Gamma, x : A, y : A, z : \mathsf{Id}[A](x;y) \vdash B\,\mathsf{type} \\ \Gamma, x : A \vdash Q : [x, x, \mathsf{refl}[A](x)/x, y, z]B\end{array}}{\Gamma \vdash \mathsf{id\text{-}ind}[A;x, y, z.B](P; x.Q) : [M, N, P/x, y, z]B}$$

$$\text{IND-BETA}\quad \frac{\begin{array}{c}\Gamma, x : A, y : A, z : \mathsf{Id}[A](x;y) \vdash B\,\mathsf{type} \\ \Gamma \vdash M : A \qquad \Gamma, x : A \vdash Q : [x, x, \mathsf{refl}[A](x)/x, y, z]B\end{array}}{\Gamma \vdash \mathsf{id\text{-}ind}[A;x, y, z.B](\mathsf{refl}[A](M); x.Q) \equiv [M/x]Q : [M, M, \mathsf{refl}[A](M)/x, y, z]B}$$

Figure 5: Identity Type

criterion the functions[5] $\lambda(x.2\times x)$ and $\lambda(x+x.)$, being not identical, or in fact *not equal* to each other! In other words it fails to equate two functions on the basis of their having the same input/output behavior, the standard notion of equality of functions in mathematics.[6] This is the infamous problem of *function extensionality* in type theory: if the identity type given in Figure 5 is used to express equality, then it fails to equate two functions with the same input/output behavior unless they are in fact the very same programs.[7]

What to do? In mathematical practice functions are invariably treated extensionally—indeed, in set theory, they are defined as sets of ordered pairs, an inherently extensional choice. But in a constructive setting they should also have computational meaning as programs. There are several extant proposals for how to manage this situation. One is simply to live with it, but that has proved to be a practical impossibility. Another is to add to type theory an axiom stating that if two functions give equal results (in the sense of the identity type) on all inputs then they are to be regarded as equal:

$$\mathit{funext} : f, g : (A \to B) \to (x : A \to \mathsf{Id}[B](\mathsf{ap}(f;x); \mathsf{ap}(g;x))) \to \mathsf{Id}[A \to B](f;g)$$

But what is the elimination rule for the identity type supposed to make of an instance of *funext*? It's only reduction property is for reflexivity, and the elimination rule amounts to the supposition that the only evidence for an identity is reflexivity. The result is a mess in which even closed elements of type bool are not simply true and false!

The fundamental problem is that equality is a *type-specific* notion, whereas the identity type axiomatizes it in a type-generic fashion. Functions are only the first symptom of the disorder. For example, if one were to have quotient types in this setting, then the fact that elements of a type are equivalent

[5] Spare the informality for the sake of argument.

[6] Indeed in set theory functions are defined to be certain sets of ordered pairs; the two indicated functions determining the same set of ordered pairs, they are identical, hence equal. But in a setting where computation matters, the representational device used in set theory is not available, it is important to have access to the *code* in order to execute it.

[7] The problem is not limited to functions, though these are often called out as the central issue; other mathematical objects fail to be equated properly when equality is formulated in this way.

must be regarded as evidence that their equivalence classes are identical, and this goes beyond mere reflexivity. There are several extant reactions to this observation. The most (in)famous, because widely criticized, though even more widely adopted for practical reasons, is to abandon the identity type in favor of the *equality* type (Martin-Löf, 1982), more on which shortly, which permits evidence for the equality of, say, two functions to suffice for them to be *judgmentally* equal, and hence interchangeable in all contexts. An alternative is to *define* equality on a type-by-type basis, with, say, equality at function types being defined to be the above-mentioned functionality principle (Altenkirch et al., 2007). A third approach is to introduce *higer-dimensional* structure in the form of "thin cubes" that codify evidence for an equation[8] and to introduce a coercion operation that enacts that evidence (Sterling et al., 2022).

All three approaches have their advantages, albeit at the expense of considerable machinery to formulate them precisely. Here we confine ourselves to the first, formulating what is called *extensional* type theory. First, to avoid possible confusion, the identity type described above is replaced by the *equality type*, $\mathsf{Eq}[A](M;N)$, with at most one inhabitant, $\star$, that witnesses the truth of an equation that is derivable judgmentally. The unicity of this element is imposed explicitly. The critical rule is the elimination rule, called *equality reflection*, which states that if there is evidence (which at most exists) for an equation, then it is judged to hold. (See 6 for specifics.) Using equality reflection one may easily prove that the identity function and the double-negation function on the booleans are judgmentally equal as elements of type $\mathsf{bool} \to \mathsf{bool}$.

The chief criticism against this formulation of equality is that by enriching the equality judgment to be mathematically sensible, it is by the same token rendered undecidable—the equality reflection rule says that to determine whether two elements of a type are equal, it suffices to search for a proof of that fact. Consequently, because typing is defined to respect equality of types, and because equality of indices of a family of types implies equality of their instances, type checking is no longer decidable. In some circles this is regarded as completely unacceptable and wrong-headed: type checking must be decidable.[9]

The other two approaches mentioned above work by encoding the influence of such evidence explicitly in the type theory, relying on one form or another of coercion to make use of it, retaining some implicit notion of equality by virtue of computation. In the case of OTT the idea is to define $\mathsf{Eq}[A](M;N)$ by induction on the structure of A; for example, the isomorphism

$$\mathsf{Eq}[A \to B](F;G) \cong x, y : A \to \mathsf{Eq}[A](x;y) \to \mathsf{Eq}[B](\mathsf{ap}(F;x);\mathsf{ap}(G;y))$$

expresses the intended extensional equality of elements of the function type. When $\mathsf{Eq}[A](M;N)$ is taken as a primitive notion as in Figure 6, the foregoing isomorphism expresses a true fact about equality at function type. Alternatively, one can *define* the equality type by induction on the structure of A, using clauses such as the one above to define the meaning of equality on a type-by-type basis. The indicated isomorphism then provides coercions back and forth that are used to mediate uses of equality in a proof. Although this summary is accurate, as far as it goes, working it out in detail is rather involved, and is not a clear "win" over the extensional approach except insofar as one insists on decidability of judgmental equality.

The last major ingredient in pure dependent type theory is that of a *universe*, a type whose elements are types. Absent universes, type theory lacks any form of "polymorphism" such as is found in many programming languages to express code whose behavior is generic across choices of types—the identity function on a type being a leading example, it requiring no knowledge of a type to simply return

[8] Or, more generally, define deformation paths within a type.

[9] Invariably ignoring the intractible complexity of the decision problem!

EQ-FORM

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : A}{\Gamma \vdash \mathsf{Eq}[A](M;N)\,\mathsf{type}}$$

EQ-INTRO

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \star : \mathsf{Eq}[A](M;M)}$$

EQ-UNICITY

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : A \qquad \Gamma \vdash P : \mathsf{Eq}[A](M;N) \qquad \Gamma \vdash Q : \mathsf{Eq}[A](M;N)}{\Gamma \vdash P \equiv Q : \mathsf{Eq}[A](M;N)}$$

EQ-REFLECTION

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : A \qquad \Gamma \vdash P : \mathsf{Eq}[A](M;N)}{\Gamma \vdash M \equiv N : A}$$

Figure 6: Equality Type

a given element of it. In their presence the genericity in type can be expressed by an ordinary function abstraction over a universe of types. Absent universes, special-purpose "large elim's," such as the conditional type expression associated with booleans discussed above, must be included in the type theory. In their presence, however, the large elimination forms are merely standard elimination forms whose result is an element of a universe. In a programming context modules are, informally, hybrid structures consisting of some types and some pieces of code, which may be construed as dependent products over a universe—and parameterized modules may be construed as dependent functions over such structures (MacQueen, 1986).

If a universe is to be a type-of-types, why not have just one, the type of *all* types, including itself? Originally Martin-Löf made this very proposal, but it was later shown by Girard to be inconsistent.[10] The standard move—since Russell—is to *stratify* universes into an infinite hierarchy, each contained within the next, with each universe closed under all type-forming operations, including formation of *prior* universes at a given level. Informally, the "union" of all of these universes determines the collection of all types, but that thought is merely conceptual, rather than codified into type theory itself.

Supporting a hierarchy of universes requires that the judgments $A\,\mathsf{type}$ and $A \equiv A'\,\mathsf{type}$ be generalized to specify a *universe level* whose structure is dictated by the requirements of predicativity (non-circularity) in the use of universes. Writing $A\,\mathsf{type}_i$ to mean that A is a type of level i, and $A \equiv A'\,\mathsf{type}_i$ to mean that A and A' are equivalent types at level i, the rules governing universes are given in Figure 7, along with the ambient rules governing the stratified type hierarchy that they induce. It is left implicit that each level in the type hierarchy is closed under the constructs discussed above, and indeed the level-free judgments can be interpreted as being schematic in the choice of level. It is a matter of convention whether the level hierarchy begins with 0 or 1, but to be fully expressive it should extend indefinitely through any finite level.

Using universes large elimination forms such as $\mathsf{If}(M;A;B)$ can be replaced by the "small" form $\mathsf{if}(M;A;B)$ whose range type is some universe $\mathcal{U}$, achieving a technical economy not available without universes.

[10] But doesn't System F quantify over all possible types as inputs? Yes, but it does not permit computing such types as *output*, which makes all the difference.

$$\textsc{U-form}\ \frac{}{\Gamma \vdash \mathcal{U}_i\ \mathsf{type}_{i+1}} \qquad \textsc{Cum}\ \frac{\Gamma \vdash A\ \mathsf{type}_i}{\Gamma \vdash A\ \mathsf{type}_{i+1}} \qquad \textsc{Cum-eq}\ \frac{\Gamma \vdash A \equiv A'\ \mathsf{type}_i}{\Gamma \vdash A \equiv A'\ \mathsf{type}_{i+1}}$$

$$\textsc{Resp}\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash A \equiv A'\ \mathsf{type}_i}{\Gamma \vdash M : A'} \qquad \textsc{Resp-eq}\ \frac{\Gamma \vdash M \doteq M' : A \qquad \Gamma \vdash A \equiv A'\ \mathsf{type}_i}{\Gamma \vdash M \doteq M' : A'}$$

$$\textsc{U-intro}\ \frac{\Gamma \vdash A\ \mathsf{type}_i}{\Gamma \vdash A : \mathcal{U}_i} \qquad \textsc{U-into-eq}\ \frac{\Gamma \vdash A \equiv A'\ \mathsf{type}_i}{\Gamma \vdash A \doteq A' : \mathcal{U}_i}$$

$$\textsc{U-elim}\ \frac{\Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash A\ \mathsf{type}_i} \qquad \textsc{U-elim-eq}\ \frac{\Gamma \vdash A \doteq A' : \mathcal{U}_i}{\Gamma \vdash A \equiv A'\ \mathsf{type}_i}$$

Figure 7: Universes

A related extension of type theory is to postulate a universe of *propositions*, which are "at most true" in that an element of a proposition is a proof of its truth, but all such proofs of a given proposition are equated. This is fundamentally a *classical* conception of proposition, exactly because it must suppress the information content of *constructive* notions of disjunction and existence. Indeed, from its inception, a characteristic of intuitionistic logic is that proofs of such propositions must determine the disjunct and exhibit a witness to the truth of an existential. One way to suppress this information is to work under Gödel's *double negation* interpretation of classical logic, whereby disjunction and existential quantification are formulated as if doubly negated, so that existence has the force "cannot fail to exist" and disjunction has the force "cannot both be false." Other, more abstract, formulations are possible, but for present purposes it is not necessary to develop these notions in detail.

3 Impure Dependent Type Theory

It is possible to incorporate effects into dependent type theory, provided that effectful computations are segregated from pure expressions. Otherwise, the role of type theory as a logic via the propositions-as-types principle is lost, and the very idea of dependency becomes questionable when, say, indices of a family of types have effects. However, it is perfectly feasible to integrate effects using methods inspired by either the lax or cbpv formulation of effects considered earlier in this course, of which the latter is developed here.

The judgmental apparatus of the theory is adjusted to account for effects by adding a class of computation types and their equality, and a class of membership and equality judgments for each computation type:

$$\textsc{susp}\ \frac{\Gamma \vdash X\ \mathsf{ctype}_i}{\Gamma \vdash \mathsf{U}(X)\ \mathsf{type}_i} \qquad \textsc{susp-eq}\ \frac{\Gamma \vdash X \equiv X'\ \mathsf{ctype}_i}{\Gamma \vdash \mathsf{U}(X) \equiv \mathsf{U}(X')\ \mathsf{type}_i}$$

$$\textsc{free}\ \frac{\Gamma \vdash A\ \mathsf{type}_i}{\Gamma \vdash \mathsf{F}(A)\ \mathsf{ctype}_i} \qquad \textsc{free-eq}\ \frac{\Gamma \vdash A \equiv A'\ \mathsf{type}_i}{\Gamma \vdash \mathsf{F}(A) \equiv \mathsf{F}(A')\ \mathsf{ctype}_i}$$

$$\textsc{pi-comp}\ \frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma, x : A \vdash X\ \mathsf{ctype}}{\Gamma \vdash x : A \rightharpoonup X\ \mathsf{ctype}}$$

Figure 8: Computation Types (Selected Rules)

1. $\Gamma \vdash X\ \mathsf{ctype}$, stating that X is a computation type in context Γ;
2. $\Gamma \vdash X \equiv X'\ \mathsf{ctype}$, stating X and X' are equal computation types in Γ;
3. $\Gamma \vdash C : X$, stating that C is a computation of type X in Γ;
4. $\Gamma \vdash C \equiv C' : X$, stating that C and C' are equal computations of type X in Γ.

Computation types must be similarly stratified to value types, with the stratification induced by the free type constructor, which includes the latter among the former.

Each level in the computation type hierarchy is closed under the constructs given in Harper (2024a), instantiated by computations as in Harper (2024c). The suspension and free type constructors are added at each level of the appropriate value and computation type hierarchy, and the partial (effectful) function type may be generalized to dependent form. See Figure 8 for a selection of rules governing these extensions in isolation from any "actual" effects. For the later, the formulations given in Harper (2024c) may be readily adapted to the computation level in the dependent setting, providing a desired integration of effects with proofs that retains the interpretation of propositions-as-types, but also permits defining and verifying properties of programs that make use of effects such as partiality, continuations, and state that are essential for "real world" programming.

In the extended setting it is natural to consider a universe of computation types, itself a value type, to permit similar forms of quantification over computation types.

Exercise 2. *Extend the value types to include a predicative hierarchy of* universes of computation types*, $\mathcal{X}\langle i\rangle$, whose elements are computation types. How does this universe relate to the universe hierarchy, $\mathcal{U}\langle i\rangle$, of value types?*

For the case of imperative programming with assignables, the computation level of type theory is a natural setting for enriching the language with *Hoare triple types* Nanevski et al. (2006); Nanevski (2024) that express pre- and post-conditions for the execution of such programs. The Iris Project Project (2024) is a richly developed system for verifying (concurrent) imperative programs in this setting.

4 Computational Semantics (Summary)

Following Martin-Löf (1982); Allen (1987); Angiuli (2019) a computational semantics for dependent type theory may be defined as follows:[11]

1. Fix a deterministic transition system $U \longmapsto U'$ for closed terms U and U', which are used to interpret the constructs of type theory. Final states are values, defined by the judgment $U\,\mathsf{val}$. For the pure fragment of type theory it is customary to consider a lazy dynamics in which values are determined by their outermost form, leaving sequencing to the impure fragment, but it is also possible to consider instead an eager formulation. Variables are considered to range over closed terms, but may in some circumstances limited to values without major alteration of the development.

2. A *type system* defines, for each predicativity level $i \geq 0$, a binary relation $A \mathrel{\dot{=}}_i A'$, called *exact type equality (at level i)*, and, for each A such that $A \mathrel{\dot{=}}_i A$, a binary relation $M \mathrel{\dot{=}}_i M' \in A$, called *exact equality of elements M, M' of type A (at level i)*. All of these relations are required to be symmetric and transitive, and closed under head expansion in all arguments. Moreover, if $A \mathrel{\dot{=}}_i A'$, then $M \mathrel{\dot{=}}_i M' \in A$ iff $M \mathrel{\dot{=}}_i M' \in A'$, and, for each $i \geq 0$, if $A \mathrel{\dot{=}}_i A'$, then $A \mathrel{\dot{=}}_{i+1} A'$ and, for $A \mathrel{\dot{=}}_i A'$, $M \mathrel{\dot{=}}_i M' \in A$ iff $M \mathrel{\dot{=}}_{i+1} M' \in A$.

3. For a type system given by the above relations, equality of contexts and of substitutions classified by valid context is defined as follows.

   (a) $\varepsilon \mathrel{\dot{=}}_i \varepsilon$, and $\emptyset \mathrel{\dot{=}}_i \emptyset \in \varepsilon$.

   (b) $x : A, \Gamma \mathrel{\dot{=}}_i x : A', \Gamma'$ iff $A \mathrel{\dot{=}}_i A'$ and, if $M \mathrel{\dot{=}}_i M' \in A$, then $[M/x]\Gamma \mathrel{\dot{=}}_i [M'/x]\Gamma'$, and $x \hookrightarrow M, \gamma \mathrel{\dot{=}}_i x \hookrightarrow M', \gamma' \in x : A, \Gamma$ iff $M \mathrel{\dot{=}}_i M' \in A$ and $\gamma \mathrel{\dot{=}}_i \gamma' \in [M/x]\Gamma$.

4. The hypothetical judgments $\Gamma \gg_i A \mathrel{\dot{=}} A'$ and $\Gamma \gg_i M \mathrel{\dot{=}} M' \in A$ are defined for semantically valid contexts and semantically valid substitutions as follows:

   (a) $\Gamma \gg_i A \mathrel{\dot{=}} A'$ iff $\gamma \mathrel{\dot{=}}_i \gamma' \in \Gamma$ implies $\hat{\gamma}(A) \mathrel{\dot{=}}_i \widehat{\gamma'}(A')$.

   (b) $\Gamma \gg_i M \mathrel{\dot{=}} M' \in A$, for $\Gamma \gg_i A \mathrel{\dot{=}} A$, iff $\gamma \mathrel{\dot{=}}_i \gamma' \in \Gamma$ implies $\hat{\gamma}(M) \mathrel{\dot{=}}_i \widehat{\gamma'}(M') \in \hat{\gamma}(A)$

Observe the dependency structure that ensures that an indexing type is defined prior to its use to define a family of types or of their elements, and that the predicativity levels are defined in order of level, requiring that these levels be *cumulative* in that one is contained in the next.

It then remains to define a variety of types and their elements with which a given type system is constructed. The following clauses define the intended definitions of exact equality at a few representative types; see Angiuli (2019) for a full development of the definitions and their properties.

- Universes internalize typing at each predicativity level:

  1. If $A, A' \Downarrow \mathcal{U}\langle i \rangle$, then $A \mathrel{\dot{=}}_{i+1} A'$.
  2. If $B \Downarrow \mathcal{U}\langle i \rangle$, then $A \mathrel{\dot{=}}_{i+1} A' \in B$ iff $A \mathrel{\dot{=}}_i A'$.

- Equality types internalize equality:

[11] Please Angiuli (2019) for a full development.

1. If $A \Downarrow \mathsf{Eq}[B](M;M')$ and $A' \Downarrow \mathsf{Eq}[B'](N;N')$, then $A \mathrel{\dot=}_i A'$ iff $B \mathrel{\dot=}_i B'$, $M \mathrel{\dot=}_i N \in B$, and $M' \mathrel{\dot=}_i N' \in B$.
2. If $A \Downarrow \mathsf{Eq}[B](N;N')$, where $A \mathrel{\dot=}_i A$, then $M \mathrel{\dot=}_i M' \in A$ iff $M, M' \Downarrow \star$ and $N \mathrel{\dot=}_i N' \in B$.

- Booleans are bits:

1. If $A, A' \Downarrow \mathsf{bool}$, then $A \mathrel{\dot=}_i A'$.
2. If $A \Downarrow \mathsf{bool}$, then $M \mathrel{\dot=}_i M' \in A$ iff $M, M' \Downarrow \mathsf{true}$ or $M, M' \Downarrow \mathsf{false}$.

- Dependent function types:

1. If $A \Downarrow x : A_1 \to A_2$ and $A' \Downarrow x : A_1' \to A_2'$, then $A \mathrel{\dot=}_i A'$ iff $A_1 \mathrel{\dot=}_i A_1'$ and, if $M \mathrel{\dot=}_i M' \in A_1$, then $[M/x]A_2 \mathrel{\dot=}_i [M'/x]A_2'$.
2. If $A \Downarrow x : A_1 \to A_2$ where $A \mathrel{\dot=}_i A$, then $M \mathrel{\dot=}_i M' \in A$ iff $M \Downarrow \lambda(x.M_2)$, $M' \Downarrow \lambda(x.M_2')$ and $M_1 \mathrel{\dot=}_i M_1' \in A_1$ implies $[M_1/x]M_2 \mathrel{\dot=}_i [M_1'/x]M_2' \in [M_1/x]A_2$.

With these formulations of exact equality in hand, it is a lengthy exercise to verify the validity of the derivable typing judgments according to the formulation given earlier.

Theorem 1 (FTLR). *1. If $\Gamma \vdash A\ \mathsf{type}_i$, then $\Gamma \gg_i A \mathrel{\dot=} A$, and if $\Gamma \vdash A \equiv A'\ \mathsf{type}_i$, then $\Gamma \gg_i A \mathrel{\dot=} A'$.*

2. If $\Gamma \vdash M : A$, where $\Gamma \gg_i A \mathrel{\dot=} A$, then $\Gamma \gg_i M \mathrel{\dot=} M \in A$, and if $\Gamma \vdash M \mathrel{\dot=} M' : A$, then $\Gamma \gg_i A \mathrel{\dot=} A$ and $\Gamma \gg_i M \mathrel{\dot=} M' \in A$.

Exercise 3. *Check that the semantic equality type may be used to interpret the identity type proposed by Martin-Löf. How is the elimination form justified in this setting?*

Exercise 4 (Thought Question). *Exercise 3 interprets the identity type as exact equality of its closed instances, an extensional interpretation. How might one define a semantics for intensional type theory that interprets the identity type as definitional equivalence?* Hint*: Such a definition must consider open terms, because definitional equivalence is defined as a congruence relating open terms.*

Accounting for effects as described in Harper (2024c) is relatively straightforward, provided that the semantics maintains the distinction between computations and expressions, linked by the modalities. There can be no general format covering all possible combinations of effects; instead, each situation is unto itself, requiring its own notion of the dynamics of computations, which is then used to give meaning to the computation types and the suspension modality, using the aforementioned notes as a guide.

The suspension type is defined in the evident manner:

1. If $A \Downarrow \mathsf{U}(X)$ and $A' \Downarrow \mathsf{U}(X')$, then $A \mathrel{\dot=}_i A'$ iff $X \mathrel{\dot=}_i X'$.
2. If $A \Downarrow \mathsf{U}(X)$, where $X \mathrel{\dot=}_i X$, then $M \mathrel{\dot=}_i M' \in A$ iff $M \Downarrow \mathsf{susp}(C)$, $M' \Downarrow \mathsf{susp}(C')$, and $C \mathrel{\dot=}_i C' \in X$.

The partial dependent function type is defined similarly to the dependent function type, albeit with the range being a computation type.

1. If $A \Downarrow x : A_1 \rightharpoonup X_2$ and $A' \Downarrow x : A_1' \rightharpoonup A_2'$, then $A \mathrel{\dot=}_i A'$ iff $A_1 \mathrel{\dot=}_i A_1'$ and if $M_1 \mathrel{\dot=}_i M_1' \in A_1$, then $[M_1/x]A_2 \mathrel{\dot=}_i [M_1'/x]A_2'$.

2. If $A \Downarrow x : A_1 \rightharpoonup X_2$, where $A \mathrel{\dot{=}}_i A$, then $\lambda(x.C_2) \mathrel{\dot{=}}_i \lambda(x.C_2') \in A$ iff $M_1 \mathrel{\dot{=}}_i M_1' \in A_1$ implies $[M_1/x]C_1 \mathrel{\dot{=}}_i [M_1'/x]C_2' \in [M_1/x]A_2$

For partial computations the requirement is that the termination behavior be the same, and that terminating computations yield exactly equal results.

1. If $A \Downarrow \mathsf{F}(B)$ and $A' \Downarrow \mathsf{F}(B')$, then $A \mathrel{\dot{=}}_i A'$ iff $B \mathrel{\dot{=}}_i B'$.

2. If $A \Downarrow \mathsf{F}(B)$, where $A \mathrel{\dot{=}}_i A$, then $C \mathrel{\dot{=}}_i C' \in A$ iff $C \Downarrow \mathsf{ret}(M)$ implies $C' \Downarrow \mathsf{ret}(M')$ and $M \mathrel{\dot{=}}_i M' \in B$, and $C' \Downarrow \mathsf{ret}(M')$ implies $C \Downarrow \mathsf{ret}[\tau](M)$ and $M \mathrel{\dot{=}}_i M' \in B$.

It is important that the possibility of non-termination be confined to computations of free type so that the crucial propositions-as-types correspondence is not disrupted. If, say, non-terminating terms were permitted, then every type would be inhabited, and hence "true" when regarded as a proposition.

Exercise 5. *Extend Theorem 1 to account for partial computations on the basis of the characterizations of the computation types given above.*

Exercise 6. *Show that the binary product of computation types, $X_1 \times X_2$, is definable from the partial function type using the boolean value type. Show that the expected β-like equational laws are derivable under this definition.* Hint: you will need to make use of a "large elimination" form for booleans mapping into computation types.

Exercise 7. *Extend the semantics of value types to include a universe of computation types as described in Exercise 2.*

Finally, it is possible to account for phase distinctions using the same methods as developed in Harper (2024b), namely to treat phases as Kripke worlds ordered logical entailment for exact equality. For example, to account for cost, the extensional phase, EXT, indicates that step-counting is to be suppressed, allowing comparison of algorithms irrespective of cost. In a dependent setting with equality types, this means that, extensionally, one may prove that insertion sort and merge sort are equal as functions on sequences, perhaps by showing that both are equal to an algorithmically "magical" function that, given a sequence, returns the sequence in sorted order. Outside of the extensional phase the cost accounting matters, allowing internalized verifications of both cost and behavior in type theory (Niu et al., 2022).

References

Stuart Frazier Allen. *A non-type-theoretic semantics for type-theoretic language*. phd, Cornell University, USA, 1987. AAI8725748.

Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–68, Freiburg Germany, October 2007. ACM. ISBN 978-1-59593-677-6. doi: 10.1145/1292597.1292608. URL `https://dl.acm.org/doi/10.1145/1292597.1292608`.

Carlo Angiuli. *Computational Semantics of Cartesian Cubical Type Theory*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, September 2019.

Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, Cambridge, England, Second edition, 2016.

Robert Harper. Call-by-push-value. Unpublished lecture note., January 2024a. URL https://www.cs.cmu.edu/~rwh/courses/atpl/pdfs/cbpv.pdf.

Robert Harper. Cost effects and phases. Unpublished lecture note., January 2024b. URL https://www.cs.cmu.edu/~rwh/courses/atpl/pdfs/cost.pdf.

Robert Harper. Effects in call-by-push-value. Unpublished lecture note., March 2024c. URL https://www.cs.cmu.edu/~rwh/courses/atpl/pdfs/effects.pdf.

David B. MacQueen. Using dependent types to express modular structure. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '86*, pages 277–286, St. Petersburg Beach, Florida, 1986. ACM Press. doi: 10.1145/512644.512670. URL http://portal.acm.org/citation.cfm?doid=512644.512670.

Per Martin-Löf. Constructive Mathematics and Computer Programming. In *Logic, Methodology and Philosophy of Science VI*, volume 104 of *Studies in Logic and Foundations of Mathematics*, pages 153–175. Elsevier, 1982. URL https://doi.org/10.1016/S0049-237X(09)70189-2.

Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*. Number 1 in Studies in proof theory Lecture notes. Bibliopolis, Napoli, 1984. ISBN 978-88-7088-105-9.

Aleks Nanevski. HTT: Hoare Type Theory, 2024. URL https://software.imdea.org/~aleks/htt/.

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and Separation in Hoare Type Theory. 2006.

Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. A cost-aware logical framework. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–31, January 2022. ISSN 2475-1421. doi: 10.1145/3498670. URL https://dl.acm.org/doi/10.1145/3498670.

Bengt Nordström, Kent Petersson, and Jan Smith. Programming in Martin-Lof's Type Theory, 1990. URL https://www.cse.chalmers.se/research/group/logic/book/.

The Iris Project. The Iris Project, 2024. URL https://iris-project.org.

Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. A Cubical Language for Bishop Sets. *Logical Methods in Computer Science*, Volume 18, Issue 1:9069, March 2022. ISSN 1860-5974. doi: 10.46298/lmcs-18(1:43)2022. URL https://lmcs.episciences.org/9069.