Computational Higher-Dimensional Type Theory

Carlo Angiuli
Carnegie Mellon University
cangiuli@cs.cmu.edu

Robert Harper
Carnegie Mellon University
rwh@cs.cmu.edu

Todd Wilson
California State University Fresno
twilson@csufresno.edu

Abstract

Formal constructive type theory has proved to be an effective language for mechanized proof. By avoiding non-constructive principles, such as the law of the excluded middle, type theory admits sharper proofs and broader interpretations of results. From a computer science perspective, interest in type theory arises from its applications to programming languages. Standard constructive type theories used in mechanization admit computational interpretations based on meta-mathematical normalization theorems. These proofs are notoriously brittle; any change to the theory potentially invalidates its computational meaning. As a case in point, Voevodsky's univalence axiom raises questions about the computational meaning of proofs.

We consider the question: *Can higher-dimensional type theory be construed as a programming language?* We answer this question affirmatively by providing a direct, deterministic operational interpretation for a representative higher-dimensional dependent type theory with higher inductive types and an instance of univalence. Rather than being a formal type theory defined by rules, it is instead a computational type theory in the sense of Martin-Löf's meaning explanations and of the NuPRL semantics. The definition of the type theory starts with programs, and defines types as specifications of program behavior. The main result is a canonicity theorem, the first of its kind, stating that closed programs of boolean type evaluate to true or false.

1. Introduction

Constructive type theory in the broadest sense is well-established as a comprehensive framework for both the analysis and implementation of programming languages, and as a comprehensive framework for mechanizing mathematics. How is it that such apparently disparate objectives could be achieved by the same framework? From the perspective of an intuitionist, it is only to be expected—the whole point of intuitionism is to achieve a synthesis of proof and program, grounding the definition of truth itself in computation. But few subscribe to such principles, and, practically speaking, constructive type theory has proved useful regardless of its ideological origins.

The key is to recognize that there are two quite different notions of constructivity in type theory, corresponding to two different conceptions of the subject. One sense, which is embodied in the formal type theories implemented in Coq [37] and Agda [30], to name two examples, is the principle of constructivity as affording *axiomatic freedom*. By avoiding commitments to non-constructive principles, such as the unrestricted law of the excluded middle, the theory is compatible with many more interpretations than it would otherwise be, including interpretations in which proofs are modeled as programs, but also other interpretations such as those derived from topology [15]. Formal type theories are defined by a collection of inference rules defining a collection of types and their inhabitants, and specifying when two such are to be considered definitionally equal [26]—essentially, when they are equal by virtue of elementary simplifications. Axioms such as the excluded middle, or other, more subtle principles that may be taken for granted (or even denied!) in classical settings may be added to express proofs that use methods outside of the constructive canon. This approach has proved hugely successful for mechanizing a broad range of mathematics while avoiding obstructive foundational arguments.

Axiomatic freedom is well and good, but the formal approach does not address the question of the applicability of type theory to programming languages. Besides being the very motivation for constructivity in the first place, we, the authors, along with many computer scientists, are interested in the application of type theory to running code. Thus, the central question for us is *what is the computational meaning of a proof*, or, in other words, *In what sense is type theory a programming language?*

From the axiomatic perspective there are two main ways of addressing this question. One is indirect: by building models in other constructive theories. To the extent that the latter have computational content, it can be transferred to type theory itself by the interpretation. The other is direct, by defining a concept of simplification, or normalization, and showing, by a substantial meta-mathematical proof that must necessarily go beyond the capabilities of the type theory itself, that every term has a unique simplest form. The operational meaning of a term is then taken to be its unique simplest form, obtained by any of a variety of means. The same idea is often used to show the decidability of type checking for the theory, which is essential because the elements of types may be considered to be formal proofs, rather than semantic constructions, and hence ought to admit decidable checking.

This account of the computational content of proofs leaves a little to be desired. For example, normalization theorems do not afford a polynomially equivalent cost model [35]. For another, there are natural concepts of computation—such as partiality, general recursive types, non-determinacy, and mutable state—that do not appear to arise from simplification of proof terms in a logical system. But perhaps these shortcomings may be remedied in the near or distant future. The more serious difficulty, in both theory and practice, is the problem of defining a type theory by rules and eliciting its computational content by metamathematics. It is never obvious how to choose the rules that define a formal type theory, particularly when one is concerned with its computational interpretation. A seemingly minor alteration can disrupt or even destroy the computational meaning of the theory when obtained by

[Copyright notice will appear here once 'preprint' option is removed.]

metamathematical analysis. Moreover, such results are notoriously brittle; a seemingly small change to the rules can entirely invalidate the established metatheory.

Two cases in point illustrate the difficulties. The first is the problem of *function extensionality*. Formal type theory includes an *identity type* of proofs that two elements of a type are equal. In the case of function types there is no proof corresponding to the principle that two functions are equal if they have the same graph. This is a serious obstacle to mechanization of mathematics, which is invariably extensional in this respect. Were computational considerations to be ignored, one could simply add an axiom of extensionality, and be done with it. But what is the computational content of such a proof? In a *tour de force* paper, Altenkirch et al. [3] showed how to elicit this content without severely disrupting the theory, but the account is rather indirect and technically involved. More recently, Voevodsky's celebrated *univalence axiom*, which equates types that are, in a technically precise sense, equivalent, raises even more serious questions. From the outset [39] a central question has been: *What is the computational content of univalence?*

Besides disrupting the standard computational interpretation of formal type theory, the univalence axiom led to consideration of the more general question of *proof-relevant* notions of equality for which the interchangeability of the sides of an equation relies on information encoded in the proof that they are equal. For example, in the case of univalence, types can be equivalent in disparate ways, so it is not immediately obvious in what sense they may be considered equal, even though doing so is standard practice in informal mathematics. Addressing this question raises the further question of when two such proofs are themselves equal. For example, one would expect the composition of equality proofs to be associative, which is to say that the two associations ought to be considered equal, which may in turn require further evidence. Such considerations are addressed by the homotopy-theoretic interpretation of constructive type theory [39] in which the previously known *identity type* is generalized to become an *identification type* whose elements are called variously *paths*, *homotopies*, or, as we shall call them here, *identifications*. The resulting theory inclusive of identifications is called *higher-dimensional type theory*. The question considered in this paper is: *Can higher-dimensional dependent type theory be regarded as a programming language, while retaining its role as a logic for mechanizing mathematics?*

We answer this question in the affirmative using a new notion of *cubical logical relations*. The use of cubical methods is inspired by the formal cubical type theories proposed by Cohen et al. [10] and Licata and Brunerie [24], which are themselves inspired by the model of homotopy type theory given by Bezem et al. [8] using Kan cubical sets, a constructive variant of a concept from classical homotopy theory. The model given by Bezem et al. [8] established the constructivity of the univalence axiom by giving an interpretation in a constructive set theory, but it does not yield in any straightforward way a direct computational interpretation of higher-dimensional dependent type theory.

The importance of cubical methods for type theory may be seen as an application of Martin-Löf's principles governing the *judgmental structure* of type theory [27, 29]. In contrast to homotopy type theory [39], cubical methods consider not only types and their elements, but also identifications among and within types, as a primitive concept. Type theory is expanded to *higher dimensions* according to the following pattern. At dimension 0 we have the usual types and their elements, which are *points* of those types. Types at dimension 1 form identifications between ordinary types, and classify identifications, or *lines*, between points. At dimension 2 are identifications between identifications, or *homotopies*, which form *squares* among types and among lines in types.

The dimensions of the universe of types and of the types themselves are populated by a variety of comprehension principles such as the aforementioned univalence axiom, and by instances of the general concept of a *higher inductive type*, which is freely generated by given cubes of arbitrary dimension. To ensure that the cubical structure expresses a sensible concept of identification, the type theory must satisfy a constructive form of the *Kan condition* similar to that of Bezem et al. [8], allowing elements to be coerced between identified types, and that they are closed under a *composition* operation that ensures, among other important criteria, that identifications are composable and reversible, and that composition is associative up to higher identification.

We elicit the computational content of higher-dimensional dependent type theory using Martin-Löf's *meaning explanations*, which are directly grounded in computation. Similar methods are used to define the NuPRL type theory [1, 38]; our work may also be seen as an extension of the NuPRL type theory to account for higher dimensions. Rather than being defined by a collection of inference rules to which a computational meaning is assigned after the fact, a meaning explanation *starts* with the notion of computation (given by a deterministic operational semantics for a collection of programs), and then *defines* types as specifications of their *behavior*. Thus, for example, a function from A to B is any program—perhaps a "foreign function" or "oracle"—that sends elements of A to elements of B. More precisely, types specify the *exact equality* of behaviors, so that, for example, functions with the same graph are exactly equal. In the higher-dimensional setting this means that we sharply distinguish exact equality (in its ordinary mathematical sense) from identification (in its homotopy-theoretic sense), rather than conflate the two as is done in other treatments of higher-dimensional structure [8, 39]. The definitions of the types are given using only constructively valid, predicative principles that would in any case have to be accepted for the metamathematical analysis of a formal type theory. Indeed, because all of the rules of higher-dimensional type theory are sound under our definition of types, and our semantics may also be seen as a realizability interpretation of formal cubical type theory, providing insight into those formalisms as well. But to view our work only as such would be to sell it short. As with the NuPRL type theory [2, 14], we view proof theory as a window on the truth, not as defining truth itself.

The main result of this paper is to answer *affirmatively* (and constructively!) the question of whether higher-dimensional type theory can be construed as a programming language. More precisely, we formulate a computational higher-dimensional dependent type theory with instances of higher inductive types and of the univalence principle, for which we can obtain the following *canonicity theorem*:

Theorem 1 (Canonicity)**.** *If* $M \in \mathsf{bool}\ [\cdot]$ *then either* $M \Downarrow \mathsf{true}$ *or* $M \Downarrow \mathsf{false}$.

As emphasized by Martin-Löf [27], the canonicity theorem confirms that theory is consistent and has a direct computational meaning. As far as we are aware, this is the first canonicity result for a higher-dimensional dependent type theory.

2. Higher-dimensional programming

Martin-Löf's meaning explanations of type theory are based on a notion of computation. In order to extend meaning explanations to higher dimension, we must first extend computation to higher dimension, by explaining how to evaluate not only ordinary terms at dimension 0, but path terms at dimension 1 and above.

In homotopy type theory [39], paths are represented as terms $P : \mathsf{Id}_A(M, N)$ of identity type; the type specifies the endpoints of a path, in this case $M, N : A$. A homotopy is a term $H : \mathsf{Id}_{\mathsf{Id}_A(M,N)}(P, Q)$ of iterated identity type. Such an iterated identity

type is not well-formed unless the two endpoints P, Q are themselves both paths with identical boundary. As the dimension increases, more and more boundary data is required to even state the type of a higher cell. This data is required not only by the judgmental apparatus, because well-formedness of a type depends on the endpoints of a path, but for the operational semantics as well, because path operations can compute endpoints.

Bezem et al. [8] suggested organizing this data with *cubical sets*, a storied mathematical idea due to Kan [23] and one of the earliest combinatorial descriptions of topological spaces. Cubical methods in type theory have since been used in applications ranging from synthetic homotopy theory to guarded recursion [9, 10, 24, 25].

(A number of related structures all go by the name "cubical sets." Experts will recognize our formulation as equivalent to presheaves on the free cartesian cube category generated by an interval object $1 \to I \leftarrow 1$, i.e., cubes with degeneracies, faces, and diagonals.)

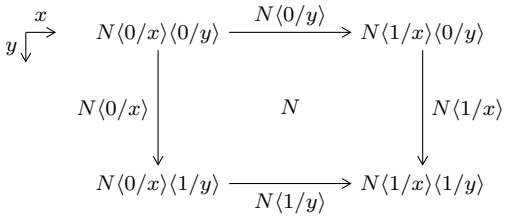
2.1 Cubical programs

Concretely, our programming language has two sorts—ordinary λ-calculus terms (written $A, B, M, N, \dots$ with variables $a, b, \dots$) and *dimension terms* (r, r'), which are either 0 or 1 (ε), or a *dimension name* ($x, y, z, \dots$). These dimension names are familiar to programming language researchers as nominal constants [31, 32] or symbols [19]. Dimension names represent formal elements of an abstract interval whose end points are notated 0 and 1.

Although dimension names look superficially like term variables, they are quite different in a few regards—we can evaluate programs with free dimension names, but not free variables, and the operational semantics compares them for equality. Importantly, however, all our constructions respect permutation of names.

Dimension terms occur as arguments to certain term formers in our programming language; for example, loop_r is a program for any dimension term r (i.e., $\mathsf{loop}_0, \mathsf{loop}_1$, and loop_x are all valid programs). We define a *dimension substitution* operation $M\langle r/x\rangle$ which replaces free occurrences of x in M with r, alongside ordinary term substitution written $M[N/a]$.

We write $\mathsf{FD}(M)$ for the set of dimension names free in M (called the *support* in nominal set literature), and say that M is a Ψ*-cube* if $\mathsf{FD}(M) \subseteq \Psi$. We call $\emptyset$-cubes *points*, x-cubes *lines*, (x, y)-cubes *squares*, and so forth. An x-cube M can be regarded it as an abstract line in the x direction, whose left endpoint, or *face*, is $M\langle 0/x\rangle$, whose right face is $M\langle 1/x\rangle$, and whose dependence on x represents the parameter x tracing out the "interior" of the line. An (x, y)-cube N is an abstract square with four lines as its boundary, and four points as the boundary of those lines:



The fact that dimension substitutions commute validates the geometrical fact that the $-\langle 0/x\rangle$ face of $N\langle 0/y\rangle$ must agree with the $-\langle 0/y\rangle$ face of $N\langle 0/x\rangle$ in the upper left.

A Ψ-cube M can be regarded trivially as a *degenerate*, or *reflexivity*, (Ψ, x)-cube whose x-faces are both M. Finally, we can substitute one dimension name for another, which takes the *diagonal* of a square. $N\langle x/y\rangle$ is an x-line (the upper-left-to-lower-right diagonal in the diagram above) whose left face is $N\langle 0/x\rangle\langle 0/y\rangle$ and whose right face is $N\langle 1/x\rangle\langle 1/y\rangle$.

We call all combinations of faces, diagonals, and degeneracies of a cube its *aspects*. Aspects are obtained by means of total dimension substitutions, written $\psi : \Psi' \to \Psi$, which take a Ψ-cube M to a Ψ'-cube $M\psi$. If we think of Ψ as a *context* of dimension names, then degeneracies correspond to weakenings, diagonals to contractions, and permutation of Ψ to exchange; these notions of aspects yield a *cartesian* notion of dimension, one quite analogous to the behavior of variables in type theory.

Note that terms are *not* to be understood solely in terms of their dimensionally-closed instances (namely, their 0-dimensional aspects). Rather, a term's dependence on dimension names is to be understood generically; geometrically, one might imagine additional unnamed points in the interior of the abstract interval.

2.2 Syntax

$$
\begin{aligned}
\text{Terms } M :=\ & (a{:}A) \to B \mid (a{:}A) \times B \mid \mathsf{Id}_{x.A}(M, N) \\
& \mid \mathsf{bool} \mid \mathsf{not}_r \mid \mathbb{S}^1 \mid \lambda a.M \mid \mathsf{app}(M, N) \mid \langle M, N\rangle \\
& \mid \mathsf{fst}(M) \mid \mathsf{snd}(M) \mid \langle x\rangle M \mid M@r \mid \mathsf{true} \mid \mathsf{false} \\
& \mid \mathsf{if}_{a.A}(M; N_1, N_2) \mid \mathsf{notel}_r(M) \mid \mathsf{base} \mid \mathsf{loop}_r \\
& \mid \mathbb{S}^1\text{-}\mathsf{elim}_{a.A}(M; N_1, x.N_2) \mid \mathsf{coe}_{x.A}^{r \rightsquigarrow r'}(M) \\
& \mid \mathsf{hcom}_A^{\overrightarrow{r_i}}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^{\varepsilon}})
\end{aligned}
$$

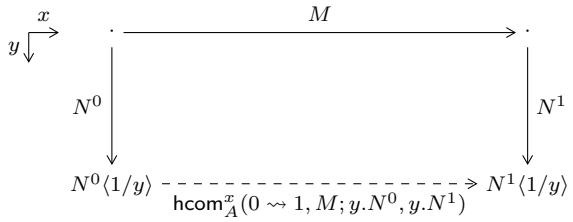
Figure 1. Term syntax.

The syntax of our cubical programming language is described in Figure 1; it is largely familiar. We write $x.-$ for dimension binders and $a.-$ for term binders. Additionally, in $(a{:}A) \to B$ and $(a{:}A) \times B$, a is bound in B. We now briefly describe the unfamiliar constructs.

$\langle x\rangle M$ is an abstraction of a dimension name x in M, and $M@r$ is application to a dimension term. $\mathbb{S}^1$ is the higher inductive type corresponding to *the circle*, which has a base point base and an x-path loop_x. Just as if takes a motive type $a.A$, a boolean M, and two cases N_1, N_2 corresponding to true and false, the elimination principle for the circle takes two cases N_1 and $x.N_2$ corresponding to its generating base point and path. not_r is an instance of the univalence principle corresponding to the negation equivalence between bool and itself; this in fact forms a type with elements $\mathsf{notel}_r(M)$, as we explain in Section 4.6.

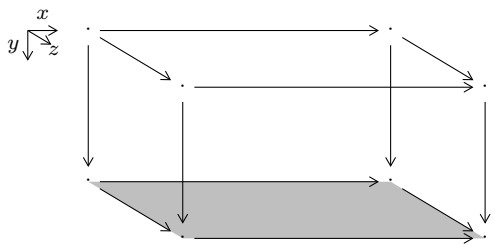
This brings us to the last major constructs of cubical type theory, namely the *Kan conditions*, coe and hcom. The first, called *coercion*, takes an x-line between types A and an element M of $A\langle r/x\rangle$ to an element of $A\langle r'/x\rangle$. This is a generalization of the *transport* operation described in the HoTT Book [39], and indeed of the ordinary notion of coercion in programming: $x.A$ is evidence identifying the types $A\langle r/x\rangle$ and $A\langle r'/x\rangle$, and coe effects the content of this identification on elements of the former type.

The second, called *homogeneous Kan composition*, states that open boxes in any type have lids. This operation endows types with a higher groupoid structure by ensuring that lines can be composed, reversed, and so forth. Some simple instances of this principle are easy to visualize.



The simplest composition scenario states that a U-shaped configuration of lines always forms the boundary to a square. If M is an x-line in A, and $y.N^\varepsilon$ are y-lines in A, and their endpoints agree as depicted above, then $\mathsf{hcom}^x_A(0 \rightsquigarrow 1, M; y.N^0, y.N^1)$ is the *composite*, an x-line in A from $N^0\langle 1/y\rangle$ to $N^1\langle 1/y\rangle$. Moreover, there is an (x, y)-square with those four lines as its boundary, namely $\mathsf{hcom}^x_A(0 \rightsquigarrow y, M; y.N^0, y.N^1)$, which is called the *filler* of this composition scenario. $(-\langle 1/y\rangle$ of the filler is clearly the composite. We will see later why the other faces agree.)

The general form of hcom is $\mathsf{hcom}^{\overrightarrow{r_i}}_A(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\varepsilon})$. M is the *cap* of the composition; r specifies which "side" the cap is on (above, the $y = 0$ side), and r' the side the composite is on (above, the $y = 1$ side). When r' is a dimension name not occurring elsewhere in the composition problem, the hcom "traces out" the interior of the composition problem, ranging from the cap to the composite, thus obtaining the filler. Finally, $\overrightarrow{r_i} = r_1, \ldots, r_n$ is a list of $n \geq 1$ dimension terms called the *extents* of the composition problem, and $\overrightarrow{y.N_i^\varepsilon}$ is a list of $2n$ *tube faces*, each with a bound dimension name. Each extent specifies the dimension across which each pair of tube faces lies; for example, the first two tube faces are on the $r_1 = 0$ and $r_1 = 1$ sides of the composition problem (above, $x = 0$ and $x = 1$). The bound dimension names in the tube faces are the dimension in which r, r' lie; binding it ensures the cap and composite do not vary in that direction.



Above is an example of a two-extent composition problem. If the top (x, z)-square is M, the left and right (y, z)-squares (across the x direction) are N^0_x, N^1_x, and the back and front (x, y)-squares (across the z direction) are N^0_z, N^1_z, then the bottom face of the cube (in gray) is an (x, z)-square, the composite

$$\mathsf{hcom}^{x,z}_A(0 \rightsquigarrow 1, M; y.N^0_x, y.N^1_x, y.N^0_z, y.N^1_z)$$

while the interior (x, y, z)-cube filler is

$$\mathsf{hcom}^{x,z}_A(0 \rightsquigarrow y, M; y.N^0_x, y.N^1_x, y.N^0_z, y.N^1_z)$$

It is difficult to depict more complex composition scenarios.

The purpose of the Kan conditions is to provide closure properties for identifications, ensuring each type satisfies the laws of higher groupoids. When we compare our theory to that of the HoTT Book [39], the Kan conditions will implement the elimination principle for the identity type. As this is the essence of finding the computational content of higher-dimensional type theory, the Kan conditions account for much of the complexity of the present work.

Our hcom operation is closely related to the uniform Kan condition [8, 16, 20] introduced by Bezem et al. [8], which differs from the ordinary Kan condition of cubical sets [23, 46] in two important ways. First, the classical definition simply states that fillers exist for all open boxes while hcom explicitly witnesses this fact, providing a composition *structure* on each type rather than simply stating a *property* of types. (Such structures have been considered in the mathematical literature on algebraic weak factorization systems [17].) Second, we do not require that the extents $\overrightarrow{r_i}$ coincide with the set of free dimension names in the term. We allow composition scenarios like the first we described, $\mathsf{hcom}^x_A(0 \rightsquigarrow 1, M; y.N^0, y.N^1)$, where the cap and tube faces depend on not only x but also z, forming a "trough" of the top, left, and right faces of a cube, composing to the bottom face. In addition, the $-\langle 0/z\rangle$ face of that bottom face must be exactly

$$\mathsf{hcom}^x_{A\langle 0/z\rangle}(0 \rightsquigarrow 1, M\langle 0/z\rangle; y.N^0\langle 0/z\rangle, y.N^1\langle 0/z\rangle)$$

This is the *uniformity* of the uniform Kan condition, which is of course very natural in syntax, because substitutions commute with term formers.

We will carefully specify the behavior of coe and hcom in Section 3.3, including the *adjacency conditions* that certain faces of the cap and tube faces must agree, as depicted above, and that certain faces of composites and fillers must agree with (faces of) the cap and tube faces.

2.3 Operational semantics

We define the operational semantics of our cubical programming language using two judgments:

1. $M\ \mathsf{val}$, stating that M is a *value*, or *canonical form*, and
2. $M \longmapsto M'$, stating that M takes *one step of evaluation* to M'.

These judgments apply to closed terms of any dimension, that is, terms containing free dimension names but not free term variables.

If $M\ \mathsf{val}$ then $M \not\longmapsto$, but the converse need not be the case. As usual, we write $M \longmapsto^* M'$ to mean that M transitions to M' in zero or more steps. We say M evaluates to V, written $M \Downarrow V$, when $M \longmapsto^* V$ and $V\ \mathsf{val}$.

Our operational semantics is deterministic, i.e.,

$$\text{If } M \longmapsto M_1 \text{ and } M \longmapsto M_2, \text{ then } M_1 = M_2.$$

up to α-equivalence. This implies a term has at most one value.

A representative sample of the operational semantics can be found in Figure 2. We have included all the rules pertaining to dependent functions and booleans, as well as the type-generic rules for the Kan conditions. (See the anonymous supplementary material for all rules.)

Most of the operational semantics rules are typical of weak head reduction: introduction forms are values, and elimination forms only evaluate their principal arguments. hcom and coe evaluate their type argument first, and have a different implementation at every type. At higher type, like $(a{:}A) \to B$, they are implemented using hcom and coe at the constituent types. At bool and $\mathbb{S}^1$, coe has no effect because the trivial coercion from bool to bool (resp., $\mathbb{S}^1$) has no effect, and hcom has the minimum implementation needed to satisfy the Kan conditions described in Section 3.3. This is because composition is primitive at base types, and higher inductive types such as bool and $\mathbb{S}^1$ are inductively generated by their constructors *and* Kan composition. (As we discuss in Section 4.1, we define bool as a higher inductive type although it has no path constructors, to demonstrate the robustness of our canonicity result.)

if evaluates its principal argument. When true it returns the first branch and when false the second; when the argument is an hcom of booleans, it returns a composition, in the motive type A,

hcom and coe

$$\frac{A \longmapsto A'}{\mathsf{coe}^{r \rightsquigarrow r'}_{x.A}(M) \longmapsto \mathsf{coe}^{r \rightsquigarrow r'}_{x.A'}(M)} \qquad \frac{A \longmapsto A'}{\mathsf{hcom}^{\overrightarrow{r_i}}_{A}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\varepsilon}) \longmapsto \mathsf{hcom}^{\overrightarrow{r_i}}_{A'}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\varepsilon})}$$

Dependent functions

$$\overline{(a{:}A) \to B\ \mathsf{val}} \qquad \frac{M \longmapsto M'}{\mathsf{app}(M, N) \longmapsto \mathsf{app}(M', N)} \qquad \overline{\mathsf{app}(\lambda a.M, N) \longmapsto M[N/a]} \qquad \overline{\lambda a.M\ \mathsf{val}}$$

$$\overline{\mathsf{hcom}^{\overrightarrow{r_i}}_{(a:A)\to B}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\varepsilon}) \longmapsto \lambda a.\mathsf{hcom}^{\overrightarrow{r_i}}_{B}(r \rightsquigarrow r', \mathsf{app}(M, a); \overrightarrow{y.\mathsf{app}(N_i^\varepsilon, a)})}$$

$$\overline{\mathsf{coe}^{r \rightsquigarrow r'}_{x.(a:A)\to B}(M) \longmapsto \lambda a.\mathsf{coe}^{r \rightsquigarrow r'}_{x.B[\mathsf{coe}^{r' \rightsquigarrow x}_{x.A}(a)/a]}(\mathsf{app}(M, \mathsf{coe}^{r' \rightsquigarrow r}_{x.A}(a)))}$$

Booleans

$$\overline{\mathsf{bool}\ \mathsf{val}} \qquad \frac{\overrightarrow{r_i} = x_1, \ldots, x_{i-1}, \varepsilon, r_{i+1}, \ldots, r_n}{\mathsf{hcom}^{\overrightarrow{r_i}}_{\mathsf{bool}}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\varepsilon}) \longmapsto N_i^\varepsilon\langle r'/y\rangle} \qquad \frac{r = r'}{\mathsf{hcom}^{x_1,\ldots,x_n}_{\mathsf{bool}}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\varepsilon}) \longmapsto M} \qquad \overline{\mathsf{true}\ \mathsf{val}} \qquad \overline{\mathsf{false}\ \mathsf{val}}$$

$$\frac{r \neq r'}{\mathsf{hcom}^{x_1,\ldots,x_n}_{\mathsf{bool}}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\varepsilon})\ \mathsf{val}} \qquad \frac{M \longmapsto M'}{\mathsf{if}_{a.A}(M; T, F) \longmapsto \mathsf{if}_{a.A}(M'; T, F)} \qquad \overline{\mathsf{if}_{a.A}(\mathsf{true}; T, F) \longmapsto T}$$

$$\overline{\mathsf{if}_{a.A}(\mathsf{false}; T, F) \longmapsto F} \qquad \frac{r \neq r' \quad H = \mathsf{hcom}^{x_1,\ldots,x_n}_{\mathsf{bool}}(r \rightsquigarrow z, M; \overrightarrow{y.N_i^\varepsilon})}{\mathsf{if}_{a.A}(\mathsf{hcom}^{x_1,\ldots,x_n}_{\mathsf{bool}}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\varepsilon}); T, F) \longmapsto \mathsf{com}^{x_1,\ldots,x_n}_{z.A[H/a]}(r \rightsquigarrow r', \mathsf{if}_{a.A}(M; T, F); \overrightarrow{y.\mathsf{if}_{a.A}(N_i^\varepsilon; T, F)})} \qquad \overline{\mathsf{coe}^{r \rightsquigarrow r'}_{x.\mathsf{bool}}(M) \longmapsto M}$$

Figure 2. Operational semantics, selected rules.

of ifs on the constituent booleans. The com term mentioned in this operational semantics rule is an abbreviation of:

$$\mathsf{com}^{\overrightarrow{r_i}}_{y.A}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\varepsilon}) := \mathsf{hcom}^{\overrightarrow{r_i}}_{A\langle r'/y\rangle}(r \rightsquigarrow r', \mathsf{coe}^{r \rightsquigarrow r'}_{y.A}(M); \overrightarrow{y.\mathsf{coe}^{y \rightsquigarrow r'}_{y.A}(N_i^\varepsilon)})$$

com implements *heterogeneous composition* by combining the two Kan operations, and is needed here because the motive A is dependent on bool, as we discuss in Section 4.1.

The primary sources of higher cubes in the syntax are the Kan conditions and the circle. The circle provides a good example of the interplay between dimension substitution and evaluation:

$$\overline{\mathsf{base}\ \mathsf{val}} \qquad \overline{\mathsf{loop}_x\ \mathsf{val}} \qquad \overline{\mathsf{loop}_\varepsilon \longmapsto \mathsf{base}}$$

Thus for any Ψ, base is a Ψ-cube and loop_x is a (Ψ, x)-cube. We arrange that the $-\langle\varepsilon/x\rangle$ faces of loop_x are base with the rule that $\mathsf{loop}_\varepsilon \longmapsto \mathsf{base}$.

Evaluating a Ψ-cube never increases its dimension, but a cube which depends on x may lose its dependency on x, thus revealing it to be degenerate. To be precise,

$$\text{If } M \longmapsto M', \text{ then } \mathsf{FD}(M') \subseteq \mathsf{FD}(M).$$

3. Cubical meaning explanations

The judgments of computational type theory are defined in terms of a system of relations between programs. In this framework, types are merely specifications of the computational behavior of these programs, and can be thought of as a system of type refinements for an untyped language. In this section, we define what is a *cubical type system*, and how it gives rise to the judgments of computational higher-dimensional type theory.

3.1 Ordinary meaning explanations

We begin by briefly recalling the ordinary meaning explanations [1, 18, 27, 38] of Martin-Löf [27], which also serve as the basis of the NuPRL type theory [1].

Computational type theory is built on two core judgments, namely that two programs are equal types

$$A \doteq B\ \mathsf{type}$$

and that two programs are equal at a type

$$M \doteq N \in A$$

The judgment A type is simply an abbreviation for $A \doteq A$ type, and $M \in A$ for $M \doteq M \in A$. To be a type is to evaluate to a *canonical type*, that is, to prescribe what it means for a canonical form to be a canonical member (or element) of that type, and for two such to be equal members of that type. To be equal types is to evaluate to canonical types equipped with identical notions of membership and equal membership. To be a member (resp., equal members) of a type A is to evaluate to a canonical member (resp., equal canonical members) of the canonical type given by the value of A. We say that A type is a *presupposition* of the judgment $M \doteq N \in A$ because we cannot make sense of this judgment until we know that A is equipped with a notion of membership.

For example, we might say that bool is a canonical type whose canonical members are true and false, such that true is equal only to true, and false to false. Then bool type, because bool $\Downarrow$ bool which is a canonical type, and $M \doteq N \in$ bool means that either $M \Downarrow$ true and $N \Downarrow$ true, or $M \Downarrow$ false and $N \Downarrow$ false.

A *type system* over a programming language is a family of relations specifying which values are (equal) canonical types, and for each canonical type, which values are (equal) canonical members of that canonical type. To be precise, a type system is

1. A symmetric and transitive relation $\approx$ over values, and
2. For each $A_0 \approx A_0$, a symmetric and transitive relation $\approx_{A_0}$ over values, such that if $A_0 \approx B_0$, the relations $\approx_{A_0}$ and $\approx_{B_0}$ are equal.

These relations are symmetric and transitive to ensure the equality judgments are as well. They are not reflexive, as not all values are canonical types and canonical members of every canonical type, but for any A_0 such that $A_0 \approx B_0$, symmetry and transitivity imply $A_0 \approx A_0$. These *partial equivalence relations* are simply a technical device for simultaneously defining a predicate and an equivalence relation on objects satisfying the predicate. A value may be a canonical member of multiple canonical types, and may be equal to another value at some types but not others.

The meanings of the core judgments are fixed by any type system: $A \doteq B$ type when $A \Downarrow A_0$, $B \Downarrow B_0$, and $A_0 \approx B_0$; and $M \doteq N \in A$ when $M \Downarrow M_0$, $N \Downarrow N_0$, $A \Downarrow A_0$, and $M_0 \approx_{A_0} N_0$.

The open judgments

$$a_1 : A_1, \ldots, a_n : A_n \gg A \doteq B \text{ type}$$
$$a_1 : A_1, \ldots, a_n : A_n \gg M \doteq N \in A$$

express that a judgment is true for all members of the types $A_1, \ldots, A_n$, and moreover that it respects the equality of those types. For example, $a : A \gg M \in B$ when for all $N \doteq N' \in A$, $M[N/a] \doteq M[N'/a] \in B[N/a]$. (The full definition is slightly involved because context types A_i can depend on earlier a_j.)

It is no mistake that we write $M \in A$ rather than $M : A$, and $\Gamma \gg M \in A$ rather than $\Gamma \vdash M : A$. The latter (perhaps more familiar) judgments of formal type theory capture the idea that M constitutes a *formal proof* of the proposition A. There, M proves a unique proposition A (up to definitional equality), and from M one can reconstruct not only A but a full derivation of $M : A$. In computational type theory, $M \doteq N \in A$ means that the programs M and N exhibit equal behaviors as specified by A. This difference surfaces in many ways:

- $M \in \mathsf{bool}$ requires only that $M \Downarrow \mathsf{true}$ or $M \Downarrow \mathsf{false}$, and nothing whatsoever about the structure of M. In contrast, $M : \mathsf{bool}$ only when M is structurally well-typed.
- $m : \mathbb{N}, n : \mathbb{N} \gg m + n \doteq n + m \in \mathbb{N}$ is true (for suitable definitions of $\mathbb{N}$ and $+$) because for any natural numbers m, n, $m+n$ and $n+m$ evaluate to the same natural number. *Proving* this fact requires an inductive argument, but it is *true* even if one cannot invent that argument. In contrast, there is no definitional equality $m : \mathbb{N}, n : \mathbb{N} \vdash m + n \equiv n + m : \mathbb{N}$, because one cannot see they are equal merely by simplifying (e.g., β-normalizing) both sides. That is, a proof about $m + n$ is not *self-evidently* a proof about $n + m$; one cannot expect to, in general, systematically reconstruct a derivation of the latter given a derivation of the former, without an induction argument.
- In computational type theory, two functions $A \to B$ are equal whenever they send equal elements of A to equal elements of B. Since we define types solely using the computational behavior of programs, there is no other sensible definition in this setting.
- For all these reasons, it is not decidable whether a program M has type A, nor should one expect it to be: $M \in A$ is an assertion about the runtime behavior of a program, not the content of a formal proof.

These differences have led some to believe that computational type theory is less useful than formal type theory. In fact, their judgments are altogether different, and suitable for different purposes! As a logic for reasoning about program behavior, the applicability of computational type theory is indisputable. On the other hand, it is quite true that $M \in A$ is not an appropriate notion of formal proof, because the structure of M may come far short of explaining why A is true. (In proof theories for computational type theory, derivations often contain much more information than M.)

3.2 Closed cubical judgments

One can regard ordinary type theory as the fragment of cubical type theory in which no terms depend on dimension names. Since cubical programs represent not only $\emptyset$-cubes but x-cubes, (x, y)-cubes, and higher, one can have canonical Ψ-cubes for any Ψ, and hence a cubical type system specifies when for any Ψ, a canonical Ψ-cube classifies other Ψ-cubes, and when two canonical Ψ-cubes are equal in such a Ψ-cube.

Definition 2. A *cubical type system* consists of

1. For every Ψ, a symmetric and transitive relation $- \approx^{\Psi} -$ over values A_0 such that $\mathsf{FD}(A_0) \subseteq \Psi$, and
2. For every $A_0 \approx^{\Psi} B_0$, symmetric and transitive relations $- \approx^{\Psi}_{A_0} -$ and $- \approx^{\Psi}_{B_0} -$ over values M_0 such that $\mathsf{FD}(M_0) \subseteq \Psi$, such that $M_0 \approx^{\Psi}_{A} N_0$ if and only if $M_0 \approx^{\Psi}_{B} N_0$.

We will sometimes write $A \sim^{\Psi} B$ when $A \Downarrow A_0$, $B \Downarrow B_0$, and $A_0 \approx^{\Psi} B_0$, and $M \sim^{\Psi}_{A} N$ when $M \Downarrow M_0$, $N \Downarrow N_0$, $A \Downarrow A_0$, and $M_0 \approx^{\Psi}_{A_0} N_0$.

The two most fundamental judgments of cubical type theory are

$$A \doteq B \text{ pretype } [\Psi] \quad \text{and} \quad M \doteq N \in A \, [\Psi]$$

Further judgments are defined in terms of these. As before, the meanings of these judgments are fixed by any cubical type system, but their definitions are more involved than simply evaluating A, B, M, N.

Complications arise because these judgments should be closed under dimension substitution. The intuitive meaning of $M \in A \, [\Psi]$ is that M is a Ψ-cube of the pretype A, so it should certainly be the case that its aspects are also cubes of the same pretype, i.e., $M\psi \in A\psi \, [\Psi']$ for all $\psi : \Psi' \to \Psi$. If we say $M \in A \, [\Psi]$ means that $A \Downarrow A_0$, $M \Downarrow M_0$, and $M_0 \approx^{\Psi}_{A_0} M_0$, there is certainly no reason why we should expect $A\psi \Downarrow A'_0$, $M\psi \Downarrow M'_0$, and $M'_0 \approx^{\Psi'}_{A'_0} M'_0$, since ψ can change the evaluation behavior of M. A concrete example is $\mathsf{hcom}^{x}_{\mathsf{bool}}(0 \rightsquigarrow 0, M; y.N, y.N)$, which steps to M but whose $-\langle 0/x \rangle$ face steps to $N\langle 0/y \rangle$.

An obvious solution is to build this condition into the meaning of the judgment, and say that $M \in A \, [\Psi]$ when for any $\psi : \Psi' \to \Psi$, $M\psi \sim^{\Psi'}_{A\psi} M\psi$. This guarantees the ψ aspect of any Ψ-cube of a pretype evaluates to a canonical Ψ'-cube of that pretype, but still says nothing about the relationship between different canonical aspects of the original Ψ-cube, or about the aspects of those canonical Ψ'-cubes. We impose the following stronger condition (where $A\psi_1\psi_2$ means $(A\psi_1)\psi_2$):

Definition 3. We say $A \doteq B$ pretype $[\Psi]$, presupposing $\mathsf{FD}(A, B) \subseteq \Psi$, when for any $\psi_1 : \Psi_1 \to \Psi$ and $\psi_2 : \Psi_2 \to \Psi_1$,

1. $A\psi_1 \Downarrow A_1$, $A_1\psi_2 \Downarrow A_2$, $A\psi_1\psi_2 \Downarrow A_{12}$,
2. $B\psi_1 \Downarrow B_1$, $B_1\psi_2 \Downarrow B_2$, $B\psi_1\psi_2 \Downarrow B_{12}$, and
3. $A_2 \approx^{\Psi_2} A_{12} \approx^{\Psi_2} B_2 \approx^{\Psi_2} B_{12}$.

Definition 4. We say $M \doteq N \in A \, [\Psi]$, presupposing $A \doteq A$ pretype $[\Psi]$ and $\mathsf{FD}(M, N) \subseteq \Psi$, when for any $\psi_1 : \Psi_1 \to \Psi$ and $\psi_2 : \Psi_2 \to \Psi_1$,

1. $M\psi_1 \Downarrow M_1$, $M_1\psi_2 \Downarrow M_2$, $M\psi_1\psi_2 \Downarrow M_{12}$,

2. $N\psi_1 \Downarrow N_1$, $N_1\psi_2 \Downarrow N_2$, $N\psi_1\psi_2 \Downarrow N_{12}$, and
3. $M_2 \approx^{\Psi_2}_{A_{12}} M_{12} \approx^{\Psi_2}_{A_{12}} N_2 \approx^{\Psi_2}_{A_{12}} N_{12}$, where $A\psi_1\psi_2 \Downarrow A_{12}$.

That is, not only do all aspects of M evaluate to canonical cubes of A, but its $\psi_1\psi_2$ aspect and the ψ_2 aspect of its ψ_1 aspect's value evaluate to equal canonical cubes of A. We say this ensures M has *coherent aspects*. Note that this coherence is not guaranteed by the operational semantics; it is the role of the type system to carve out the sensible programs.

As before, we write A $\mathsf{pretype}$ $[\Psi]$ when $A \doteq A$ $\mathsf{pretype}$ $[\Psi]$, and $M \in A$ $[\Psi]$ when $M \doteq M \in A$ $[\Psi]$. (We will continue to abbreviate judgments in this fashion without further comment.) The judgments $A \doteq B$ $\mathsf{pretype}$ $[\Psi]$ and $M \doteq N \in A$ $[\Psi]$ are symmetric and transitive, so if $A \doteq B$ $\mathsf{pretype}$ $[\Psi]$ then A $\mathsf{pretype}$ $[\Psi]$ and B $\mathsf{pretype}$ $[\Psi]$, and if $M \doteq N \in A$ $[\Psi]$ then $M \in A$ $[\Psi]$ and $N \in A$ $[\Psi]$. Equal pretypes have equal elements, i.e., if $A \doteq B$ $\mathsf{pretype}$ $[\Psi]$ and $M \doteq N \in A$ $[\Psi]$, then $M \doteq N \in B$ $[\Psi]$.

The coherent aspect condition tells us that in a sense, any pair of dimension substitutions commute with evaluation. If a pretype is *cubical*, then the values of any of its cubes' aspects again have coherent aspects, and hence arbitrary interleavings of dimension substitutions and evaluation are coherent, i.e., the act of applying a dimension substitution and then evaluating is functorial.

Definition 5. We say A $\mathsf{pretype}$ $[\Psi]$ is *cubical* if for any $\psi : \Psi' \to \Psi$ and $M \approx^{\Psi'}_{A_0} N$ (where $A\psi \Downarrow A_0$), $M \doteq N \in A\psi$ $[\Psi']$.

If we consider only terms M with no free dimension names, then $M = M\psi$ and the cubical meaning explanations essentially coincide with the ordinary ones: $A \doteq B$ $\mathsf{pretype}$ $[\Psi]$ when $A \sim^{\Psi'} B$ for all Ψ', and $M \doteq N \in A$ $[\Psi]$ when $M \sim^{\Psi'}_A N$ for all Ψ'.

3.3 Context restrictions and Kan conditions

So far, we have explained what a cubical *pretype* is. A *type* is a cubical pretype which is moreover *Kan*, meaning that it is closed under the hcom and coe operations in a certain sense. Specifying the hcom operation in particular is quite challenging, because a composition scenario is valid in a pretype not only when the cap and tube faces are elements of that pretype, but when certain faces of the cap and tube faces are equal elements.

These equalities are called the *adjacency conditions*, and are implicit in the diagrams in Section 2.2. For example, in the U-shaped configuration, $M\langle\varepsilon/x\rangle$ and $N^\varepsilon\langle\varepsilon/y\rangle$ must be equal points, while in the box-shaped configuration, each tube face intersects the cap in a line, and each pair of adjacent tube faces must also intersect, for a total of eight equalities.

There are a few more subtleties to address. An extent r_i can be 0 or 1, in addition to being a dimension name. Indeed, if the composite of the U, $\mathsf{hcom}^x_A(0 \rightsquigarrow 1, M; y.N^0, y.N^1)$, is meant to be an x-line from $N^0\langle 1/y\rangle$ to $N^1\langle 1/y\rangle$, then its $-\langle 0/x\rangle$ face,

$$\mathsf{hcom}^0_{A\langle 0/x\rangle}(0 \rightsquigarrow 1, M\langle 0/x\rangle; y.N^0\langle 0/x\rangle, y.N^1\langle 0/x\rangle)$$

must be a $\emptyset$-cube member of $A\langle 0/x\rangle$ equal to $N^0\langle 1/y\rangle$. Next, although we depicted N^ε as y-lines and not (x, y)-squares, there is no reason why the extent x cannot occur in them. Rather than prohibit this syntactically, we say that any x occurring in N^ε should be interpreted as an ε instead. Lastly, two extents r_i, r_j can be equal dimension names, in which case two different pairs of tube faces are exactly superpositioned, and must be equal.

These constraints are all handled elegantly by *dimension context restrictions*, which are sets of unoriented equations $\Xi = (r_1 = r'_1, \ldots, r_n = r'_n)$ in Ψ. We say that $\psi : \Psi' \to \Psi$ *satisfies* Ξ if for each $i \in [1, n]$, either $r_i\psi = r'_i\psi = 0$, $r_i\psi = r'_i\psi = 1$, or $r_i\psi = r'_i\psi = x$. Then:

Definition 6. We say $A \doteq B$ $\mathsf{pretype}$ $[\Psi \mid \Xi]$, presupposing $\mathsf{FD}(A, B, \Xi) \subseteq \Psi$, when for any $\psi : \Psi' \to \Psi$ satisfying Ξ, $A\psi \doteq B\psi$ $\mathsf{pretype}$ $[\Psi']$.

Definition 7. We say $M \doteq N \in A$ $[\Psi \mid \Xi]$, presupposing $\mathsf{FD}(M, N, A, \Xi) \subseteq \Psi$ and A $\mathsf{pretype}$ $[\Psi \mid \Xi]$, when for any $\psi : \Psi' \to \Psi$ satisfying Ξ, $M\psi \doteq N\psi \in A\psi$ $[\Psi']$.

The intuition is that Ξ restricts the ordinary judgment only to certain faces of the specified terms. Here are some examples, writing $\mathcal{J}$ for either judgment:

- $\mathcal{J}$ $[\Psi \mid \cdot]$ iff $\mathcal{J}$ $[\Psi]$. By definition, since all ψ satisfy an empty set of equations, $\mathcal{J}$ $[\Psi \mid \cdot]$ iff $\mathcal{J}\psi$ $[\Psi']$ for all $\psi : \Psi' \to \Psi$. But $\mathcal{J}\psi$ $[\Psi']$ for all ψ iff $\mathcal{J}$ $[\Psi]$, because the judgments are closed under dimension substitution.
- $\mathcal{J}$ $[\Psi, x \mid x = 0, x = 1]$ always, because no ψ can satisfy $x\psi = 0$ and $x\psi = 1$ simultaneously.
- $\mathcal{J}$ $[\Psi, x, y \mid x = 0, y = 1]$ iff $\mathcal{J}\langle 0/x\rangle\langle 1/y\rangle$ $[\Psi]$, because all dimension substitutions satisfying $x\psi = 0$ and $y\psi = 1$ can be written as a composition of $-\langle 0/x\rangle\langle 1/y\rangle$ and another dimension substitution.

Definition 8. We say A, B are *equally Kan*, presupposing $A \doteq B$ $\mathsf{pretype}$ $[\Psi]$, if the following five conditions hold:

1. For any $\psi : \Psi' \to \Psi$, if
   (a) $M \doteq O \in A\psi$ $[\Psi']$,
   (b) $N^\varepsilon_i \doteq N^{\varepsilon'}_j \in A\psi$ $[\Psi', y \mid r_i = \varepsilon, r_j = \varepsilon']$ for any $i \in [1, n]$, $j \in [1, n]$, $\varepsilon = 0, 1$, and $\varepsilon' = 0, 1$,
   (c) $N^\varepsilon_i \doteq P^\varepsilon_i \in A\psi$ $[\Psi', y \mid r_i = \varepsilon]$ for any $i \in [1, n]$ and $\varepsilon = 0, 1$, and
   (d) $N^\varepsilon_i\langle r/y\rangle \doteq M \in A\psi$ $[\Psi' \mid r_i = \varepsilon]$ for any $i \in [1, n]$ and $\varepsilon = 0, 1$,
   then $\mathsf{hcom}^{\overrightarrow{r_i}}_{A\psi}(r \rightsquigarrow r', M; \overrightarrow{y.N^\varepsilon_i}) \doteq \mathsf{hcom}^{\overrightarrow{r_i}}_{B\psi}(r \rightsquigarrow r', O; \overrightarrow{y.P^\varepsilon_i}) \in A\psi$ $[\Psi']$.
2. For any $\psi : \Psi' \to \Psi$, if
   (a) $M \in A\psi$ $[\Psi']$,
   (b) $N^\varepsilon_i \doteq N^{\varepsilon'}_j \in A\psi$ $[\Psi', y \mid r_i = \varepsilon, r_j = \varepsilon']$ for any $i \in [1, n]$, $j \in [1, n]$, $\varepsilon = 0, 1$, and $\varepsilon' = 0, 1$, and
   (c) $N^\varepsilon_i\langle r/y\rangle \doteq M \in A\psi$ $[\Psi' \mid r_i = \varepsilon]$ for any $i \in [1, n]$ and $\varepsilon = 0, 1$,
   then $\mathsf{hcom}^{\overrightarrow{r_i}}_{A\psi}(r \rightsquigarrow r, M; \overrightarrow{y.N^\varepsilon_i}) \doteq M \in A\psi$ $[\Psi']$.
3. For any $\psi : \Psi' \to \Psi$, under the same conditions as above, if $r_i = \varepsilon$ for some i then
   $$\mathsf{hcom}^{\overrightarrow{r_i}}_{A\psi}(r \rightsquigarrow r', M; \overrightarrow{y.N^\varepsilon_i}) \doteq N^\varepsilon_i\langle r'/y\rangle \in A\psi\ [\Psi'].$$
4. For any $\psi : (\Psi', x) \to \Psi$, if $M \doteq N \in A\psi\langle r/x\rangle$ $[\Psi']$, then $\mathsf{coe}^{r \rightsquigarrow r'}_{x.A\psi}(M) \doteq \mathsf{coe}^{r \rightsquigarrow r'}_{x.B\psi}(N) \in A\psi\langle r'/x\rangle$ $[\Psi']$.
5. For any $\psi : (\Psi', x) \to \Psi$, if $M \in A\psi\langle r/x\rangle$ $[\Psi']$, then $\mathsf{coe}^{r \rightsquigarrow r}_{x.A\psi}(M) \doteq M \in A\psi\langle r/x\rangle$ $[\Psi']$.

Definition 9. We say $A \doteq B$ type $[\Psi]$, presupposing $A \doteq B$ $\mathsf{pretype}$ $[\Psi]$, when A and B are cubical and equally Kan.

The first Kan condition states that hcom is an element of a type whenever (a) the cap M is an element, (b) the ε side of the r_i tube is equal to the ε' side of the r_j tube for all $i, j, \varepsilon, \varepsilon'$, and (d) the ε side of the r_i tube agrees on its $-\langle r/y\rangle$ side with the $r_i = \varepsilon$ side of the cap; and moreover, hcom respects equality in all arguments including the type subscript. If $r_i = x$, then by (b) when $i = j$ and $\varepsilon = \varepsilon'$,

$$N^\varepsilon_i \doteq N^\varepsilon_i \in A\psi\ [\Psi', y \mid x = \varepsilon, x = \varepsilon]$$

which is to say that the entirety of the tube face N^ε_i is an element, where any occurrences of x are replaced by ε. And when $i = j$ but

ε and ε' are opposite faces,

$$N_i^0 \doteq N_i^1 \in A\psi\ [\Psi', y \mid x = 0, x = 1]$$

which is to say there is no adjacency condition between a tube face and its opposite tube face. If $r_i = 0$, then we have

$$N_i^0 \doteq N_i^0 \in A\psi\ [\Psi', y \mid 0 = 0, 0 = 0]$$
$$N_i^1 \doteq N_i^1 \in A\psi\ [\Psi', y \mid 0 = 1, 0 = 1]$$

which is to say that the 0 side of the r_i tube is an element, and there are no conditions on the 1 side.

The second Kan condition states that when $r = r'$ and hcom is an element, it is equal to its cap M. This condition ensures the filler of the U, $\mathsf{hcom}^x_A(0 \rightsquigarrow y, M; y.N^0, y.N^1)$, has M as its $-\langle 0/y\rangle$ face. The third condition states that when $r_i = \varepsilon$ and hcom is an element, it is equal to $N_i^\varepsilon\langle r'/y\rangle$. This condition ensures the $-\langle\varepsilon/x\rangle$ faces of the filler of the U are N^ε, and the $-\langle\varepsilon/x\rangle$ faces of the composite are $N_i^\varepsilon\langle 1/y\rangle$.

The fourth and fifth Kan conditions state that coe along $x.A$ sends elements of $A\langle r/x\rangle$ to elements of $A\langle r'/x\rangle$, and that when $r = r'$ it has no effect. All five conditions quantify over ψ to ensure that being Kan, and hence being a type, is closed under dimension substitution. Being equally Kan is symmetric and transitive, hence being a type is as well.

The Kan conditions utilize very specific context restrictions, so the added machinery is not, strictly speaking, necessary. One could instead spell out what the context restrictions mean for every $r_i, r_j, \varepsilon, \varepsilon'$, but such a definition would be difficult to state or use.

The hcom operation performs *homogeneous* composition, in the sense that A cannot depend on y, the dimension in which r, r' lie and the tube faces are bound. By combining it with coe, one can obtain a *heterogeneous* composition operation com, whose definition was given in Section 2.3. The analogue of the first Kan condition is given below; analogues of the second and third Kan conditions hold as well.

Theorem 10. *If* $A \doteq B\ \mathsf{type}\ [\Psi]$, $\psi : (\Psi', y) \to \Psi$,

1. $M \doteq O \in A\psi\langle r/y\rangle\ [\Psi']$,
2. $N_i^\varepsilon \doteq N_j^{\varepsilon'} \in A\psi\ [\Psi', y \mid r_i = \varepsilon, r_j = \varepsilon']$ *for any* $i, j, \varepsilon, \varepsilon'$,
3. $N_i^\varepsilon \doteq P_i^\varepsilon \in A\psi\ [\Psi', y \mid r_i = \varepsilon]$ *for any* i, ε,
4. $N_i^\varepsilon\langle r/y\rangle \doteq M \in A\psi\langle r/y\rangle\ [\Psi' \mid r_i = \varepsilon]$ *for any* i, ε,

then $\mathsf{com}^{\overrightarrow{r_i}}_{y.A\psi}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\varepsilon}) \doteq \mathsf{com}^{\overrightarrow{r_i}}_{y.B\psi}(r \rightsquigarrow r', O; \overrightarrow{y.P_i^\varepsilon}) \in A\psi\langle r'/y\rangle\ [\Psi']$.

3.4 Open cubical judgments

As before, the open judgments express the truth of a judgment universally over members of $A_1, \ldots, A_n$. The Ψ specifies at which dimension the context, pretype, and elements are initially considered, but one must additionally require the quantification over the context to hold at all aspects $A_1\psi, \ldots, A_n\psi$. The open judgments are defined mutually, stratified by the length of the context.

Definition 11. We say $(a_1 : A_1, \ldots, a_n : A_n)\ \mathsf{ctx}\ [\Psi]$ when

$$A_1\ \mathsf{pretype}\ [\Psi],$$
$$a_1 : A_1 \gg A_2\ \mathsf{pretype}\ [\Psi], \ldots$$
$$\text{and } a_1 : A_1, \ldots, a_{n-1} : A_{n-1} \gg A_n\ \mathsf{pretype}\ [\Psi].$$

Definition 12. We say $a_1 : A_1, \ldots, a_n : A_n \gg B \doteq B'\ \mathsf{pretype}\ [\Psi]$, presupposing $(a_1 : A_1, \ldots, a_n : A_n)\ \mathsf{ctx}\ [\Psi]$, when for any $\psi : \Psi' \to \Psi$ and any

$$N_1 \doteq N_1' \in A_1\psi\ [\Psi'],$$
$$N_2 \doteq N_2' \in A_2\psi[N_1/a_1]\ [\Psi'], \ldots$$
$$\text{and } N_n \doteq N_n' \in A_n\psi[N_1, \ldots, N_{n-1}/a_1, \ldots, a_n]\ [\Psi'], \text{ we have}$$
$$B\psi[N_1, \ldots, N_n/a_1, \ldots, a_n]$$
$$\doteq B'\psi[N_1', \ldots, N_n'/a_1, \ldots, a_n]\ \mathsf{pretype}\ [\Psi'].$$

Definition 13. We say $a_1 : A_1, \ldots, a_n : A_n \gg M \doteq M' \in B\ [\Psi]$, presupposing $a_1 : A_1, \ldots, a_n : A_n \gg B\ \mathsf{pretype}\ [\Psi]$, when for any $\psi : \Psi' \to \Psi$ and any

$$N_1 \doteq N_1' \in A_1\psi\ [\Psi'],$$
$$N_2 \doteq N_2' \in A_2\psi[N_1/a_1]\ [\Psi'], \ldots$$
$$\text{and } N_n \doteq N_n' \in A_n\psi[N_1, \ldots, N_{n-1}/a_1, \ldots, a_n]\ [\Psi'], \text{ we have}$$
$$M\psi[N_1, \ldots, N_n/a_1, \ldots, a_n] \doteq M'\psi[N_1', \ldots, N_n'/a_1, \ldots, a_n]$$
$$\in B\psi[N_1, \ldots, N_n/a_1, \ldots, a_n]\ [\Psi'].$$

We say $\Gamma \gg B \doteq B'\ \mathsf{type}\ [\Psi]$, presupposing $\Gamma \gg B \doteq B'\ \mathsf{pretype}\ [\Psi]$, when for any equal members of the context types, the corresponding instances of B, B' are equal closed types. We define context-restricted open judgments $\Gamma \gg \mathcal{J}\ [\Psi \mid \Xi]$ to mean that $\Gamma\psi \gg \mathcal{J}\psi\ [\Psi']$ for any $\psi : \Psi' \to \Psi$ satisfying Ξ. Open analogues of the Kan conditions, and of Theorem 10, hold for equal open types.

Like the closed judgments, these judgments are all symmetric, transitive, and closed under dimension substitution. The earlier hypotheses in each definition ensure that later hypotheses are sensible; for example, $(a_1 : A_1, \ldots, a_n : A_n)\ \mathsf{ctx}\ [\Psi]$ and $N_1 \in A_1\psi\ [\Psi']$ ensure that $A_2\psi[N_1/a_1]\ \mathsf{pretype}\ [\Psi']$.

Since types are programs, and open judgments are given meaning by substitution, dependency is simply a consequence of allowing types to contain variables. It is natural that types can depend on dimension names, because if $a : A \gg B\ \mathsf{type}\ [\cdot]$ and M is an x-line of A, then $B[M/a]$ is a type varying in x, with endpoints $B[M\langle\varepsilon/x\rangle/a]$.

4. Types

Now that we have defined the judgments of computational cubical type theory, we can consider what it means for a cubical type system to have a type of booleans, of dependent functions, and so forth. (In general, a cubical type system need not have any types, which is not particularly interesting!) For each type former, we prove that any cubical type system closed under it satisfies typing rules (formation, introduction, elimination, computation, etc.) similar to those found in the formal cubical type theories of Cohen et al. [10] and Licata and Brunerie [24]. (All proofs mentioned in this section can be found in the anonymous supplementary material.)

One can interpret this work as an extensional realizability model of a formal cubical type theory, by modeling the latter judgments by our computational cubical judgments. However, we hope to take full advantage of our computational judgments to consider higher-dimensional analogues of concepts like exact equality [1], partiality [13], and strict subsets [11] previously considered in NuPRL.

A selection of rules validated in our model can be found in Figures 3 and 4. Figure 3 contains rules valid in *all* cubical type systems, including various structural rules, the Kan conditions, and "restriction rules" for deriving the hypotheses of the Kan conditions. Figure 4 contains rules valid in any cubical type system closed under dependent functions, dependent pairs, identifications, booleans, the circle, and not_x. (Again, we emphasize that these rules do not *define* our theory, but are rather theorems about particular cubical type theories.) For the sake of concision and clarity, these rules are stated in *local form*, extending them to *global form* by *uniformity*, also called *naturality*. (This format was suggested by Martin-Löf [28], itself inspired by Gentzen's original concept of natural deduction.) In some rules for dependent func-

Structural rules

$$\frac{A\ \mathsf{type}\ [\Psi]}{a : A \gg a \in A\ [\Psi]} \qquad \frac{\mathcal{J}\ [\Psi] \quad A\ \mathsf{type}\ [\Psi]}{a : A \gg \mathcal{J}\ [\Psi]} \qquad \frac{\mathcal{J}\ [\Psi] \quad \psi : \Psi' \to \Psi}{\mathcal{J}\psi\ [\Psi']} \qquad \frac{A \doteq A'\ \mathsf{type}\ [\Psi]}{A' \doteq A\ \mathsf{type}\ [\Psi]} \qquad \frac{A \doteq A'\ \mathsf{type}\ [\Psi] \quad A' \doteq A''\ \mathsf{type}\ [\Psi]}{A \doteq A''\ \mathsf{type}\ [\Psi]}$$

$$\frac{M' \doteq M \in A\ [\Psi]}{M \doteq M' \in A\ [\Psi]} \qquad \frac{M \doteq M' \in A\ [\Psi] \quad M' \doteq M'' \in A\ [\Psi]}{M \doteq M'' \in A\ [\Psi]} \qquad \frac{M \doteq M' \in A\ [\Psi] \quad A \doteq A'\ \mathsf{type}\ [\Psi]}{M \doteq M' \in A'\ [\Psi]}$$

$$\frac{a : A \gg B \doteq B'\ \mathsf{type}\ [\Psi] \quad N \doteq N' \in A\ [\Psi]}{B[N/a] \doteq B'[N'/a]\ \mathsf{type}\ [\Psi]} \qquad \frac{a : A \gg M \doteq M' \in B\ [\Psi] \quad N \doteq N' \in A\ [\Psi]}{M[N/a] \doteq M'[N'/a] \in B[N/a]\ [\Psi]}$$

Context restrictions

$$\frac{\mathcal{J}\ [\Psi]}{\mathcal{J}\ [\Psi \mid \cdot]} \qquad \frac{\mathcal{J}\ [\Psi \mid \Xi]}{\mathcal{J}\ [\Psi \mid \Xi, r = r]} \qquad \frac{}{\mathcal{J}\ [\Psi \mid \Xi, 0 = 1]} \qquad \frac{}{\mathcal{J}\ [\Psi, x \mid x = 0, x = 1]} \qquad \frac{\mathcal{J}\langle \varepsilon/x\rangle\ [\Psi]}{\mathcal{J}\ [\Psi, x \mid x = \varepsilon]} \qquad \frac{\mathcal{J}\langle \varepsilon/x\rangle\langle \varepsilon'/y\rangle\ [\Psi]}{\mathcal{J}\ [\Psi, x, y \mid x = \varepsilon, y = \varepsilon']}$$

Kan operations

$$\frac{\begin{array}{ll} & A \doteq A'\ \mathsf{type}\ [\Psi] \\ & M \doteq O \in A\ [\Psi] \\ (\forall i, \varepsilon) & N_i^\varepsilon \doteq P_i^\varepsilon \in A\ [\Psi, y \mid r_i = \varepsilon] \\ (\forall i, j, \varepsilon, \varepsilon') & N_i^\varepsilon \doteq N_j^{\varepsilon'} \in A\ [\Psi, y \mid r_i = \varepsilon, r_j = \varepsilon'] \\ (\forall i, \varepsilon) & N_i^\varepsilon\langle r/y\rangle \doteq M \in A\ [\Psi \mid r_i = \varepsilon] \end{array}}{\mathsf{hcom}_A^{\overrightarrow{r_i}}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\varepsilon}) \doteq \mathsf{hcom}_{A'}^{\overrightarrow{r_i}}(r \rightsquigarrow r', O; \overrightarrow{y.P_i^\varepsilon}) \in A\ [\Psi]} \qquad \frac{\begin{array}{ll} & A\ \mathsf{type}\ [\Psi] \\ & M \in A\ [\Psi] \\ (\forall i, j, \varepsilon, \varepsilon') & N_i^\varepsilon \doteq N_j^{\varepsilon'} \in A\ [\Psi, y \mid r_i = \varepsilon, r_j = \varepsilon'] \\ (\forall i, \varepsilon) & N_i^\varepsilon\langle r/y\rangle \doteq M \in A\ [\Psi \mid r_i = \varepsilon] \end{array}}{\mathsf{hcom}_A^{\overrightarrow{r_i}}(r \rightsquigarrow r, M; \overrightarrow{y.N_i^\varepsilon}) \doteq M \in A\ [\Psi]}$$

$$\frac{\begin{array}{ll} & A\ \mathsf{type}\ [\Psi] \\ & M \in A\ [\Psi] \\ (\forall i, j, \varepsilon, \varepsilon') & N_i^\varepsilon \doteq N_j^{\varepsilon'} \in A\ [\Psi, y \mid r_i = \varepsilon, r_j = \varepsilon'] \\ (\forall i, \varepsilon) & N_i^\varepsilon\langle r/y\rangle \doteq M \in A\ [\Psi \mid r_i = \varepsilon] \end{array}}{\mathsf{hcom}_A^{r_1, \ldots, r_{i-1}, \varepsilon, r_{i+1}, \ldots, r_n}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\varepsilon}) \doteq N_i^\varepsilon\langle r'/y\rangle \in A\ [\Psi]}$$

$$\frac{A \doteq A'\ \mathsf{type}\ [\Psi, x] \quad M \doteq N \in A\langle r/x\rangle\ [\Psi]}{\mathsf{coe}_{x.A}^{r \rightsquigarrow r'}(M) \doteq \mathsf{coe}_{x.A'}^{r \rightsquigarrow r'}(N) \in A\langle r'/x\rangle\ [\Psi]} \qquad \frac{A\ \mathsf{type}\ [\Psi, x] \quad M \in A\langle r/x\rangle\ [\Psi]}{\mathsf{coe}_{x.A}^{r \rightsquigarrow r}(M) \doteq M \in A\langle r/x\rangle\ [\Psi]}$$

Figure 3. Rules valid in all cubical type systems.

tion and pair types we have suppressed the hypotheses $A\ \mathsf{type}\ [\Psi]$ and $a : A \gg B\ \mathsf{type}\ [\Psi]$; and for identification types, the hypothesis $A\ \mathsf{type}\ [\Psi, x]$.

Once we have explained when a cubical type system is closed under the type formers of interest, we still need to prove some such cubical type system exists. A straightforward method is to produce the *smallest* such cubical type system, by means of a fixed point construction [1, 18] starting from a cubical type theory with base types, and adjoining all new dependent function, pair, and identification types at each step. These extensions are *monotone* in the sense that they do not affect the meanings of any prior types.

Note, however, that *any* cubical type system closed under the type formers is sufficient for our purposes; none of our theorems hold only in the least such. It is therefore possible to extend our results with additional type formers (such as universes, full univalence, or more higher inductive types) without affecting the present constructions.

4.1 Booleans

A cubical type system *has booleans* if $\mathsf{bool} \approx^\Psi \mathsf{bool}$ for all Ψ, and $- \approx^-_{\mathsf{bool}} -$ is the least relation such that:

1. $\mathsf{true} \approx^\Psi_{\mathsf{bool}} \mathsf{true}$,
2. $\mathsf{false} \approx^\Psi_{\mathsf{bool}} \mathsf{false}$, and
3. $\mathsf{hcom}^{\overrightarrow{x_i}}_{\mathsf{bool}}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\varepsilon}) \approx^\Psi_{\mathsf{bool}} \mathsf{hcom}^{\overrightarrow{x_i}}_{\mathsf{bool}}(r \rightsquigarrow r', O; \overrightarrow{y.P_i^\varepsilon})$ whenever $r \neq r'$,
   (a) $M \doteq O \in \mathsf{bool}\ [\Psi]$,
   (b) $N_i^\varepsilon \doteq N_j^{\varepsilon'} \in \mathsf{bool}\ [\Psi, y \mid x_i = \varepsilon, x_j = \varepsilon']$ for all $i, j, \varepsilon, \varepsilon'$,
   (c) $N_i^\varepsilon \doteq P_i^\varepsilon \in \mathsf{bool}\ [\Psi, y \mid x_i = \varepsilon]$ for all i, ε, and
   (d) $N_i^\varepsilon\langle r/y\rangle \doteq M \in \mathsf{bool}\ [\Psi \mid x_i = \varepsilon]$ for all i, ε.

Note that $- \approx^\Psi_{\mathsf{bool}} -$ is symmetric because in the third case, (a) and (c) imply that (b) and (d) also hold for P_i^ε and O. This definition is mutual across all Ψ, which is visible after one expands the definitions of the equality judgments.

The canonicity theorem (if $M \in \mathsf{bool}\ [\cdot]$ then $M \Downarrow \mathsf{true}$ or $M \Downarrow \mathsf{false}$) holds trivially by construction, since $M \in \mathsf{bool}\ [\cdot]$ implies $M \Downarrow M_0$ and $M_0 \approx^\cdot_{\mathsf{bool}} M_0$, and the hcom case is impossible in an empty dimension context. Consistency (that $\mathsf{true} \doteq \mathsf{false} \in \mathsf{bool}\ [\Psi]$ does not hold) follows trivially as well.

The first two cases ensure that true and false are $\emptyset$-cubes of bool, and in fact Ψ-cubes (degenerately) for all Ψ. The third case ensures that bool is Kan by freely adding certain Kan composites as higher cubes. The operational semantics for hcom in bool (Figure 2) state that any hcom with an ε extent evaluates to a face of the corresponding tube face (for the first such extent), and any hcom with $r = r'$ evaluates to its cap. These ensure the second

Dependent function types

$$\frac{A \doteq A' \;\mathsf{type}\;[\Psi] \quad a : A \gg B \doteq B' \;\mathsf{type}\;[\Psi]}{(a{:}A) \to B \doteq (a{:}A') \to B' \;\mathsf{type}\;[\Psi]} \qquad \frac{a : A \gg M \doteq M' \in B\;[\Psi]}{\lambda a.M \doteq \lambda a.M' \in (a{:}A) \to B\;[\Psi]} \qquad \frac{M \doteq M' \in (a{:}A) \to B\;[\Psi] \quad N \doteq N' \in A\;[\Psi]}{\mathsf{app}(M,N) \doteq \mathsf{app}(M',N') \in B[N/a]\;[\Psi]}$$

$$\frac{a : A \gg M \in B\;[\Psi] \quad N \in A\;[\Psi]}{\mathsf{app}(\lambda a.M, N) \doteq M[N/a] \in B[N/a]\;[\Psi]} \qquad \frac{M \in (a{:}A) \to B\;[\Psi]}{M \doteq \lambda a.\mathsf{app}(M,a) \in (a{:}A) \to B\;[\Psi]}$$

Dependent pair types

$$\frac{A \doteq A' \;\mathsf{type}\;[\Psi] \quad a : A \gg B \doteq B' \;\mathsf{type}\;[\Psi]}{(a{:}A) \times B \doteq (a{:}A') \times B' \;\mathsf{type}\;[\Psi]} \qquad \frac{M \doteq M' \in A\;[\Psi] \quad N \doteq N' \in B[M/a]\;[\Psi]}{\langle M,N\rangle \doteq \langle M',N'\rangle \in (a{:}A) \times B\;[\Psi]} \qquad \frac{P \doteq P' \in (a{:}A) \times B\;[\Psi]}{\mathsf{fst}(P) \doteq \mathsf{fst}(P') \in A\;[\Psi]}$$

$$\frac{P \doteq P' \in (a{:}A) \times B\;[\Psi]}{\mathsf{snd}(P) \doteq \mathsf{snd}(P') \in B[\mathsf{fst}(P)/a]\;[\Psi]} \qquad \frac{M \in A\;[\Psi] \quad N \in B[M/a]\;[\Psi]}{\mathsf{fst}(\langle M,N\rangle) \doteq M \in A\;[\Psi]} \qquad \frac{M \in A\;[\Psi] \quad N \in B[M/a]\;[\Psi]}{\mathsf{snd}(\langle M,N\rangle) \doteq N \in B[M/a]\;[\Psi]}$$

$$\frac{P \in (a{:}A) \times B\;[\Psi]}{P \doteq \langle \mathsf{fst}(P), \mathsf{snd}(P)\rangle \in (a{:}A) \times B\;[\Psi]}$$

Identification types

$$\frac{A \doteq A' \;\mathsf{type}\;[\Psi,x] \quad P_0 \doteq P_0' \in A\langle 0/x\rangle\;[\Psi] \quad P_1 \doteq P_1' \in A\langle 1/x\rangle\;[\Psi]}{\mathsf{Id}_{x.A}(P_0,P_1) \doteq \mathsf{Id}_{x.A'}(P_0',P_1') \;\mathsf{type}\;[\Psi]}$$

$$\frac{M \doteq M' \in A\;[\Psi,x] \quad M\langle 0/x\rangle \doteq P_0 \in A\langle 0/x\rangle\;[\Psi] \quad M\langle 1/x\rangle \doteq P_1 \in A\langle 1/x\rangle\;[\Psi]}{\langle x\rangle M \doteq \langle x\rangle M' \in \mathsf{Id}_{x.A}(P_0,P_1)\;[\Psi]} \qquad \frac{M \doteq M' \in \mathsf{Id}_{x.A}(P_0,P_1)\;[\Psi]}{M@r \doteq M'@r \in A\langle r/x\rangle\;[\Psi]}$$

$$\frac{M \in \mathsf{Id}_{x.A}(P_0,P_1)\;[\Psi]}{M@\varepsilon \doteq P_\varepsilon \in A\langle \varepsilon/x\rangle\;[\Psi]} \qquad \frac{M \in A\;[\Psi,x]}{(\langle x\rangle M)@r \doteq M\langle r/x\rangle \in A\langle r/x\rangle\;[\Psi]} \qquad \frac{M \in \mathsf{Id}_{x.A}(P_0,P_1)\;[\Psi]}{M \doteq \langle x\rangle(M@x) \in \mathsf{Id}_{x.A}(P_0,P_1)\;[\Psi]}$$

Booleans

$$\frac{}{\mathsf{bool}\;\mathsf{type}\;[\Psi]} \qquad \frac{}{\mathsf{true} \in \mathsf{bool}\;[\Psi]} \qquad \frac{}{\mathsf{false} \in \mathsf{bool}\;[\Psi]}$$

$$\frac{a : \mathsf{bool} \gg A \doteq A' \;\mathsf{type}\;[\Psi] \quad M \doteq M' \in \mathsf{bool}\;[\Psi] \quad T \doteq T' \in A[\mathsf{true}/a]\;[\Psi] \quad F \doteq F' \in A[\mathsf{false}/a]\;[\Psi]}{\mathsf{if}_{a.A}(M;T,F) \doteq \mathsf{if}_{a.A'}(M';T',F') \in A[M/a]\;[\Psi]}$$

$$\frac{a : \mathsf{bool} \gg A \;\mathsf{type}\;[\Psi] \quad T \in A[\mathsf{true}/a]\;[\Psi] \quad F \in A[\mathsf{false}/a]\;[\Psi]}{\mathsf{if}_{a.A}(\mathsf{true};T,F) \doteq T \in A[\mathsf{true}/a]\;[\Psi]} \qquad \frac{a : \mathsf{bool} \gg A \;\mathsf{type}\;[\Psi] \quad T \in A[\mathsf{true}/a]\;[\Psi] \quad F \in A[\mathsf{false}/a]\;[\Psi]}{\mathsf{if}_{a.A}(\mathsf{false};T,F) \doteq F \in A[\mathsf{false}/a]\;[\Psi]}$$

Circle

$$\frac{}{\mathbb{S}^1\;\mathsf{type}\;[\Psi]} \qquad \frac{}{\mathsf{base} \in \mathbb{S}^1\;[\Psi]} \qquad \frac{}{\mathsf{loop}_r \in \mathbb{S}^1\;[\Psi]} \qquad \frac{}{\mathsf{loop}_\varepsilon \doteq \mathsf{base} \in \mathbb{S}^1\;[\Psi]}$$

$$\frac{\begin{array}{cc} a : \mathbb{S}^1 \gg A \doteq A' \;\mathsf{type}\;[\Psi] & M \doteq M' \in \mathbb{S}^1\;[\Psi] \\ P \doteq P' \in A[\mathsf{base}/a]\;[\Psi] \qquad L \doteq L' \in A[\mathsf{loop}_x/a]\;[\Psi,x] & (\forall \varepsilon)\; L\langle \varepsilon/x\rangle \doteq P \in A[\mathsf{base}/a]\;[\Psi] \end{array}}{\mathbb{S}^1\text{-}\mathsf{elim}_{a.A}(M;P,x.L) \doteq \mathbb{S}^1\text{-}\mathsf{elim}_{a.A'}(M';P',x.L') \in A[M/a]\;[\Psi]}$$

$$\frac{a : \mathbb{S}^1 \gg A \;\mathsf{type}\;[\Psi] \quad L \in A[\mathsf{loop}_x/a]\;[\Psi,x] \quad (\forall \varepsilon)\; L\langle \varepsilon/x\rangle \doteq P \in A[\mathsf{base}/a]\;[\Psi]}{\mathbb{S}^1\text{-}\mathsf{elim}_{a.A}(\mathsf{base};P,x.L) \doteq P \in A[\mathsf{base}/a]\;[\Psi]}$$

$$\frac{a : \mathbb{S}^1 \gg A \;\mathsf{type}\;[\Psi] \quad L \in A[\mathsf{loop}_x/a]\;[\Psi,x] \quad (\forall \varepsilon)\; L\langle \varepsilon/x\rangle \doteq P \in A[\mathsf{base}/a]\;[\Psi]}{\mathbb{S}^1\text{-}\mathsf{elim}_{a.A}(\mathsf{loop}_r;P,x.L) \doteq L\langle r/x\rangle \in A[\mathsf{loop}_r/a]\;[\Psi]}$$

not_x

$$\frac{}{\mathsf{not}_r\;\mathsf{type}\;[\Psi]} \qquad \frac{}{\mathsf{not}_\varepsilon \doteq \mathsf{bool}\;\mathsf{type}\;[\Psi]} \qquad \frac{M \in \mathsf{bool}\;[\Psi]}{\mathsf{coe}^{0\rightsquigarrow 1}_{x.\mathsf{not}_x}(M) \doteq \mathsf{not}(M) \in \mathsf{bool}\;[\Psi]} \qquad \frac{M \in \mathsf{bool}\;[\Psi]}{\mathsf{coe}^{1\rightsquigarrow 0}_{x.\mathsf{not}_x}(M) \doteq \mathsf{not}(M) \in \mathsf{bool}\;[\Psi]}$$

Figure 4. Rules valid in cubical type systems with the appropriate type formers.

and third Kan conditions, respectively. The remaining hcoms, those with $r \neq r'$ and all extents x_i, are irreducible and are canonical Ψ-cubes of bool to ensure the first Kan condition.

That the operational semantics examines the extents from left to right and before r and r' is an arbitrary choice to ensure determinacy. If both $r_i = \varepsilon$ and $r = r'$, for example, then the hcom steps to $N_i^\varepsilon\langle r'/y\rangle$ but by the second Kan condition it must be equal to M. This holds because, by adjacency condition (d), $N_i^\varepsilon\langle r/y\rangle$ and M are themselves equal.

Since bool has no path constructors, we could alternatively define its canonical Ψ-cubes as only true and false, and have composites of true evaluate to true, and so forth. We opt to make its composites canonical as in any higher inductive type, to demonstrate the robustness of our canonicity theorem and our treatment of not_x. In practice, however, it may be convenient to define (instead or in addition) a type of "strict booleans."

We prove that closed versions of the formation, introduction, elimination, and computation rules located in Figure 4 hold in any cubical type system with booleans. The formation rule $\mathsf{bool}\ \mathsf{type}\ [\Psi]$ requires bool to be a pretype, Kan, and cubical. The generalizations of these rules to open elements follows by the definition of the open judgments, the respect for equality built into the introduction and elimination rules, and the fact that dimension and term substitutions commute with the term formers.

When eliminating into $a : \mathsf{bool} \gg A\ \mathsf{type}\ [\Psi]$ and the principal argument evaluates to $\mathsf{hcom}^{\overrightarrow{x_i}}_{\mathsf{bool}}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\varepsilon})$, we would like to step this to a composition of $\mathsf{if}_{a.A}(M; T, F)$ and $\mathsf{if}_{a.A}(N_i^\varepsilon; T, F)$ in A, but these have different types $A[M/a]$ and $A[N_i^\varepsilon/a]$ respectively. The solution is to perform a heterogeneous composition in $A[H/a]$, where H is the filler for the above hcom, a type which has $A[M/a]$ and $A[N_i^\varepsilon/a]$ as its faces.

4.2 Circle

A cubical type system *has the circle* if $\mathbb{S}^1 \approx^\Psi \mathbb{S}^1$ for all Ψ, and $- \approx^-_{\mathbb{S}^1} -$ is the least relation such that:

1. $\mathsf{base} \approx^\Psi_{\mathbb{S}^1} \mathsf{base}$,
2. $\mathsf{loop}_x \approx^\Psi_{\mathbb{S}^1} \mathsf{loop}_x$, and
3. $\mathsf{hcom}^{\overrightarrow{x_i}}_{\mathbb{S}^1}(r \rightsquigarrow r', M; \overrightarrow{y.N_i^\varepsilon}) \approx^\Psi_{\mathbb{S}^1} \mathsf{hcom}^{\overrightarrow{x_i}}_{\mathbb{S}^1}(r \rightsquigarrow r', O; \overrightarrow{y.P_i^\varepsilon})$ whenever $r \neq r'$,
   (a) $M \doteq O \in \mathbb{S}^1\ [\Psi]$,
   (b) $N_i^\varepsilon \doteq N_j^{\varepsilon'} \in \mathbb{S}^1\ [\Psi, y \mid x_i = \varepsilon, x_j = \varepsilon']$ for all $i, j, \varepsilon, \varepsilon'$,
   (c) $N_i^\varepsilon \doteq P_i^\varepsilon \in \mathbb{S}^1\ [\Psi, y \mid x_i = \varepsilon]$ for all i, ε, and
   (d) $N_i^\varepsilon\langle r/y\rangle \doteq M \in \mathbb{S}^1\ [\Psi \mid x_i = \varepsilon]$ for all i, ε.

This is very similar to our definition of having booleans, except that $\mathbb{S}^1$ has a path constructor loop_x.

If a cubical type system has the circle, then the formation, introduction, elimination, and computation rules in Figure 4 hold. To eliminate into $a : \mathbb{S}^1 \gg A\ \mathsf{type}\ [\Psi]$, one must provide a point $P \in A[\mathsf{base}/a]\ [\Psi]$ and a loop $L \in A[\mathsf{loop}_x/a]\ [\Psi, x]$ with endpoints P.

4.3 Dependent functions

If a cubical type system has $A \doteq A'\ \mathsf{type}\ [\Psi]$ and $a : A \gg B \doteq B'\ \mathsf{type}\ [\Psi]$, we say it also *has their dependent function type* when for all $\psi : \Psi' \to \Psi$, $(a{:}A\psi) \to B\psi \approx^{\Psi'} (a{:}A'\psi) \to B'\psi$, and $- \approx^{\Psi'}_{(a:A\psi)\to B\psi} -$ is the least relation such that

$$\lambda a.M \approx^{\Psi'}_{(a:A\psi)\to B\psi} \lambda a.M'$$

when $a : A\psi \gg M \doteq M' \in B\psi\ [\Psi']$. A cubical type system is *closed under dependent functions* if it has all dependent function types in this sense.

Recalling the meaning of the open judgments, this essentially states that a Ψ-cube (resp., equal Ψ-cubes) of $(a{:}A) \to B$ is a program that evaluates to a lambda whose body sends equal elements $N \doteq N' \in A\psi\ [\Psi']$ to equal elements of $B\psi[N/a]$.

The Kan operations on $(a{:}A) \to B$ are implemented in the operational semantics (Figure 2) using the Kan operations of A and (instances of) B. For example, coercing a function from r to r' works by coercing its argument backwards from r' to r, applying the function, and coercing the result forward from r to r'.

If a cubical type system has $A \doteq A'\ \mathsf{type}\ [\Psi]$, $a : A \gg B \doteq B'\ \mathsf{type}\ [\Psi]$, and their dependent function type, then the formation, introduction, elimination, computation, and eta rules in Figure 4 hold.

4.4 Dependent pairs

If a cubical type system has $A \doteq A'\ \mathsf{type}\ [\Psi]$ and $a : A \gg B \doteq B'\ \mathsf{type}\ [\Psi]$, we say it also *has their dependent pair type* when for all $\psi : \Psi' \to \Psi$, $(a{:}A\psi) \times B\psi \approx^{\Psi'} (a{:}A'\psi) \times B'\psi$, and $- \approx^{\Psi'}_{(a:A\psi)\times B\psi} -$ is the least relation such that

$$\langle M, N\rangle \approx^{\Psi'}_{(a:A\psi)\times B\psi} \langle M', N'\rangle$$

when $M \doteq M' \in A\psi\ [\Psi']$ and $N \doteq N' \in B\psi[M/a]\ [\Psi']$. A cubical type system is *closed under dependent pairs* if it has all dependent pair types in this sense.

If a cubical type system has $A \doteq A'\ \mathsf{type}\ [\Psi]$, $a : A \gg B \doteq B'\ \mathsf{type}\ [\Psi]$, and their dependent pair type, then the formation, introduction, elimination, computation, and eta rules in Figure 4 hold.

4.5 Identification types

If a cubical type system has $A \doteq A'\ \mathsf{type}\ [\Psi, x]$, $P_0 \doteq P_0' \in A\langle 0/x\rangle\ [\Psi]$, and $P_1 \doteq P_1' \in A\langle 1/x\rangle\ [\Psi]$, we say it also *has their identification type* when for all $\psi : \Psi' \to \Psi$, $\mathsf{Id}_{x.A\psi}(P_0\psi, P_1\psi) \approx^{\Psi'} \mathsf{Id}_{x.A'\psi}(P_0'\psi, P_1'\psi)$, and $- \approx^{\Psi'}_{\mathsf{Id}_{x.A\psi}(P_0\psi, P_1\psi)} -$ is the least relation such that

$$\langle x\rangle M \approx^{\Psi'}_{\mathsf{Id}_{x.A\psi}(P_0\psi, P_1\psi)} \langle x\rangle M'$$

when $M \doteq M' \in A\psi\ [\Psi', x]$ and $M\langle\varepsilon/x\rangle \doteq P_\varepsilon\psi \in A\psi\langle\varepsilon/x\rangle\ [\Psi']$ for all ε. A cubical type system is *closed under identifications* if it has all identification types in this sense.

Ψ-cubes of $\mathsf{Id}_{x.A}(P_0, P_1)$ are thus programs that evaluate to abstracted (Ψ, x)-cubes of A whose $-\langle\varepsilon/x\rangle$ faces are P_ε. Hence the identification type simply captures the notion of a dimension shift. Kan composition in $\mathsf{Id}_{x.A}(P_0, P_1)$ performs composition in A, adding P_0, P_1 as an additional set of tube faces to constrain the faces of the composite.

If a cubical type system has $A \doteq A'\ \mathsf{type}\ [\Psi, x]$, $P_0 \doteq P_0' \in A\langle 0/x\rangle\ [\Psi]$, $P_1 \doteq P_1' \in A\langle 1/x\rangle\ [\Psi]$, and their identification type, then the formation, introduction, elimination, computation, and eta rules in Figure 4 hold.

This cubical formulation of identification types is a significant departure from the identity type of the HoTT Book [39]. These types can be related, however. For example, for any $M \in A\ [\Psi]$ we have a reflexivity identification $\langle_\rangle M \in \mathsf{Id}_{_.A}(M, M)\ [\Psi]$; for any type family $a : A \gg B\ \mathsf{type}\ [\Psi]$, identification $P \in \mathsf{Id}_{_.A}(P_0, P_1)\ [\Psi]$, and element $M \in B[P_0/a]\ [\Psi]$ of the type family at the left endpoint we have a *transport* of that element to the right endpoint: $\mathsf{coe}^{0\rightsquigarrow 1}_{x.B[P@x/a]}(M) \in B[P_1/a]\ [\Psi]$.

4.6 Not

If a cubical type system has booleans, we say it *has the* not_x *type* when $\mathsf{not}_x \approx^{\Psi,x} \mathsf{not}_x$ for all Ψ, and $- \approx^{\Psi,x}_{\mathsf{not}_x} -$ is the least relation such that:

$$\mathsf{notel}_x(M) \approx^{\Psi,x}_{\mathsf{not}_x} \mathsf{notel}_x(M')$$

when $M \doteq M' \in \mathsf{bool}\ [\Psi, x]$.

not_x is an x-line between bool and itself; the difference between not_x and bool is that when $M \in \mathsf{bool}\ [\Psi]$,

$$\mathsf{coe}^{1 \rightsquigarrow 0}_{x.\mathsf{not}_x}(M) \doteq \mathsf{not}(M) \in \mathsf{bool}\ [\Psi]$$

where $\mathsf{not}(-) := \mathsf{if}_{_.\mathsf{bool}}(-; \mathsf{false}, \mathsf{true})$, whereas

$$\mathsf{coe}^{1 \rightsquigarrow 0}_{x.\mathsf{bool}}(M) \doteq M \in \mathsf{bool}\ [\Psi].$$

not_x is an instance of univalence, for the equivalence $\mathsf{not}(-)$.

In ordinary homotopy type theory, univalence does not have elements, but this is inevitable in cubical type theory because $\mathsf{coe}^{1 \rightsquigarrow x}_{x.\mathsf{not}_x}(M) \in \mathsf{not}_x\ [\Psi, x]$ is an x-line between $\mathsf{not}(M\langle 0/x\rangle)$ and $M\langle 1/x\rangle$; in fact, this line evaluates to $\mathsf{notel}_x(M)$.

If a cubical type system has booleans and not_x, the rules in Figure 4 hold.

5. Related and Future Work

Many authors contributed to the discovery of higher-dimensional structure in type theory and its relation to homotopy theory. The key contributors to these discoveries include Awodey and Warren [7], Hofmann and Streicher [21], van den Berg and Garner [40], Voevodsky [41], Warren [45]. Many ideas were consolidated and advanced during a year-long program at the IAS in Princeton, much of which is documented in the HoTT Book [39], and which is being carried forward in the UniMath Project [42, 44]. Apart from the over-arching influence of Martin-Löf and Constable, the most direct influences on the present work are the cubical model of homotopy type theory given by Bezem et al. [8], and the formal cubical type theories introduced by Licata and Brunerie [24] and Cohen et al. [10]. Whereas our approach makes use of the cartesian cube category suggested by the former of these, the latter has been developed further by Cohen et al. [10] using a more sophisticated algebraic structure of cubes. The full technical details of the results in this paper are given in Angiuli and Harper [4], Angiuli et al. [6].

This paper represents the first step in the development of computational higher-dimensional type theory; much remains to be done. Just as logical relations enable reasoning about higher-order constructs in programming languages, our cubical logical relations are an important first step toward understanding the role of higher dimensions. While programming applications of higher-dimensional type theory remain largely unexplored, developing a concrete operational semantics is essential for both future and pre-existing work in that area, such as the implementations of patch theories as higher inductive types given by Angiuli et al. [5].

One important direction for future work is to develop a computational account of the full univalence axiom, which states that there is an equivalence between the equivalences and identifications between types in the universe. From our point of view the univalence principle is not tied to a universe *per se*, but to the very concept of a type—we consider that universes are restrictions of the "multiverse" of all types to evade Girard's paradox.

Another important direction is to develop further typing constructs in the higher-dimensional setting, including the numerous ideas developed in NuPRL, including partial types [13], inductive types [12], and subset types [11], and to consider programming constructs such as general recursion [13], bar recursion [34], and exceptions [33]. Another direction is to develop a proof theory and implementation for the type theory considered here. The nascent RedPRL system [36] seems a suitable vehicle for such explorations.

Finding useful proof theories for higher type theory poses some challenges. The formal cubical type theory as proposed by Licata and Brunerie [24] is sound for the meaning explanations given here, and may be a good start. More ambitiously, it would be interesting to develop methods for handling both exact equality and identification in the same setting, which requires a proof theory that can deal with both pretypes and types. Voevodsky's HTS type theory [43] provides some useful initial ideas. All of these extensions, especially the exact equality type, could prove helpful when mechanizing synthetic homotopy theory.

It is also important to investigate the relationship between the present theory and the formal cubical type theory given by Cohen et al. [10]. Their notion of a "cube" is technically different from ours, using an algebraic structure that is not present in our setting. It would be useful to understand in what sense our semantics is sound for their theory. Conversely, it might also be interesting to view our work as a metamathematical device for analyzing their system. We are aware [22] that Simon Huber is working to establish a canonicity theorem for their system; it would be interesting to compare our work to such a proof. As has historically been the case prior to the development of higher type theory, there is ample room for exploration of the relationship between and the practical use of both the computational and formal approaches to type theory.

Acknowledgments

The authors gratefully acknowledge the support of the Air Force Office of Scientific Research through MURI grant FA9550-15-1-0053. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFOSR.

References

[1] S. F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.

[2] S. F. Allen, M. Bickford, R. L. Constable, R. Eaton, C. Kreitz, L. Lorigo, and E. Moran. Innovations in computational type theory using Nuprl. *Journal of Applied Logic*, 4(4):428–469, 2006.

[3] T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now. In *Programming Languages meets Program Verification Workshop*, 2007.

[4] C. Angiuli and R. Harper. Computational higher type theory II: Dependent cubical realizability. Preprint, June 2016. URL `http://arxiv.org/abs/1606.09638`.

[5] C. Angiuli, E. Morehouse, D. R. Licata, and R. Harper. Homotopical patch theory. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 243–256, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2873-9. doi: 10.1145/2628136.2628158. URL `http://doi.acm.org/10.1145/2628136.2628158`.

[6] C. Angiuli, R. Harper, and T. Wilson. Computational higher type theory I: Abstract cubical realizability. Preprint, April 2016. URL `http://arxiv.org/abs/1604.08873`.

[7] S. Awodey and M. A. Warren. Homotopy theoretic models of identity types. *Mathematical Proceedings of the Cambridge Philosophical Society*, 146(1):45–55, Jan. 2009. ISSN 0305-0041. doi: 10.1017/S0305004108001783.

[8] M. Bezem, T. Coquand, and S. Huber. A model of type theory in cubical sets. In *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, volume 26, pages 107–128, 2014.

[9] L. Birkedal, A. Bizjak, R. Clouston, H. B. Grathwohl, B. Spitters, and A. Vezzosi. Guarded cubical type theory: Path equality for guarded recursion. Preprint, June 2016. URL `https://arxiv.org/abs/1606.05223`.

[10] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. (To appear), January 2016.

[11] R. L. Constable. Constructive mathematics as a programming logic I: Some principles of theory. In *Annals of Mathematics*, volume 24, pages 21–37. Elsevier Science Publishers, B.V. (North-Holland), 1985. Reprinted from *Topics in the Theory of Computation*, Selected Papers of the International Conference on Foundations of Computation Theory, FCT '83.

[12] R. L. Constable and N. P. Mendler. Recursive definitions in type theory. In *Proceedings of the Logics of Programming Conference*, pages 61–78, Jan. 1985. Cornell TR 85–659.

[13] R. L. Constable and S. F. Smith. Computational foundations of basic recursive function theory. *Theoretical Comput. Sci.*, 121(1&2):89–112, Dec. 1993.

[14] R. L. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development Environment*. Prentice-Hall, 1985.

[15] M. Escardo and T. Streicher. The intrinsic topology of martin-löf universes. To appear, *Annals of Pure and Applied Logic*, Feb. 2016.

[16] N. Gambino and C. Sattler. Uniform fibrations and the Frobenius condition. Preprint, Oct 2015. URL `http://arxiv.org/abs/1510.00669`.

[17] M. Grandis and W. Tholen. Natural weak factorization systems. *Archivum Mathematicum*, 042(4):397–408, 2006. URL `https://www.emis.de/journals/AM/06-4/tholen.pdf`.

[18] R. Harper. Constructing type systems over an operational semantics. *J. Symb. Comput.*, 14(1):71–84, July 1992. ISSN 0747-7171. doi: 10.1016/0747-7171(92)90026-Z. URL `http://dx.doi.org/10.1016/0747-7171(92)90026-Z`.

[19] R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, second edition, 2016.

[20] R. Harper and K.-B. (Favonia) Hou. A note on the uniform Kan condition in nominal cubical sets. Preprint, Jan 2015. URL `http://arxiv.org/abs/1501.05691`.

[21] M. Hofmann and T. Streicher. The groupoid interpretation of type theory. In *Twenty-five years of constructive type theory*. Oxford University Press, 1998.

[22] S. Huber. Canonicity for cubical type theory. (Private communication), March 2016.

[23] D. M. Kan. Abstract homotopy. i. *Proceedings of the National Academy of Sciences of the United States of America*, 41(12):1092–1096, 1955. ISSN 00278424. URL `http://www.jstor.org/stable/89108`.

[24] D. R. Licata and G. Brunerie. A cubical type theory, November 2014. URL `http://dlicata.web.wesleyan.edu/pubs/lb14cubical/lb14cubes-oxford.pdf`. Talk at Oxford Homotopy Type Theory Workshop.

[25] D. R. Licata and G. Brunerie. A cubical approach to synthetic homotopy theory. In *Proceedings of the 2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, LICS '15, pages 92–103, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4799-8875-4. doi: 10.1109/LICS.2015.19. URL `http://dx.doi.org/10.1109/LICS.2015.19`.

[26] P. Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In S. Kanger, editor, *Proceedings of the Third Scandinavian Logic Symposium*, volume 82 of *Studies in Logic and the Foundations of Mathematics*, pages 81 – 109. Elsevier, 1975. doi: http://dx.doi.org/10.1016/S0049-237X(08)70727-4. URL `http://www.sciencedirect.com/science/article/pii/S0049237X08707274`.

[27] P. Martin-Löf. Constructive mathematics and computer programming. *Philosophical Transactions of the Royal Society of London Series A*, 312:501–518, Oct. 1984. doi: 10.1098/rsta.1984.0073.

[28] P. Martin-Löf. Intuitionistic type theory. *Naples: Bibliopolis*, 1984.

[29] P. Martin-Löf. Verificationism then and now. In M. van der Schaar, editor, *Judgement and the Epistemic Foundation of Logic*, volume 31 of *Logic, Epistemology, and the Unity of Science*, pages 3–14. Springer, 2013. ISBN 978-94-007-5136-1. doi: 10.1007/978-94-007-5137-8_1. URL `http://dx.doi.org/10.1007/978-94-007-5137-8_1`.

[30] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[31] A. M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*, volume 57 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2013. ISBN 9781107017788.

[32] A. M. Pitts. Nominal Presentation of Cubical Sets Models of Type Theory. In H. Herbelin, P. Letouzey, and M. Sozeau, editors, *20th International Conference on Types for Proofs and Programs (TYPES 2014)*, volume 39 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 202–220, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-88-0. doi: http://dx.doi.org/10.4230/LIPIcs.TYPES.2014.202. URL `http://drops.dagstuhl.de/opus/volltexte/2015/5498`.

[33] V. Rahli and M. Bickford. A nominal exploration of intuitionism. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, pages 130–141, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4127-1. doi: 10.1145/2854065.2854077. URL `http://doi.acm.org/10.1145/2854065.2854077`.

[34] V. Rahli and M. Bickford. Coq as a Metatheory for Nuprl with Bar Induction. Presented at *Continuity, Computability, Constructivity – From Logic to Algorithms (CCC 2015)*, Sept 14, 2015.

[35] M. Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[36] J. Sterling, D. Gratzer, V. Rahli, D. Morrison, E. Akentyev, and A. Tosun. RedPRL – the People's Refinement Logic. `http://www.redprl.org/`, 2016.

[37] The Coq Project. The Coq proof assistant, 2016. URL `http://www.coq.inria.fr`.

[38] The NuPRL Project. Prl project, 2016. URL `http://nuprl.org`.

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

[40] B. van den Berg and R. Garner. Types are weak ω-groupoids. *Proceedings of the London Mathematical Society*, 102(2):370–394, 2011.

[41] V. Voevodsky. A very short note on homotopy λ-calculus, 09 2006. URL `http://www.math.ias.edu/vladimir/files/2006_09_Hlambda.pdf`.

[42] V. Voevodsky. Univalent foundations of mathematics. In *Logic, Language, Information and Computation - 18th International Workshop, WoLLIC 2011, Philadelphia, PA, USA, May 18-20, 2011. Proceedings*, page 4, 2011. doi: 10.1007/978-3-642-20920-8_4. URL `http://dx.doi.org/10.1007/978-3-642-20920-8_4`.

[43] V. Voevodsky. A simple type system with two identity types. Lecture notes, Feb. 2013. URL `https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/HTS.pdf`.

[44] V. Voevodsky. The UniMath project, 2016. URL `https://github.com/UniMath/`.

[45] M. A. Warren. *Homotopy theoretic aspects of constructive type theory*. PhD thesis, Carnegie Mellon University, 2008. URL `http://mawarren.net/papers/phd.pdf`.

[46] R. Williamson. Combinatorial homotopy theory. Lecture notes, Oct 2012. URL `http://rwilliamson-mathematics.info/combinatorial_homotopy_theory.pdf`.