Foundations and Applications of Higher-Dimensional Directed Type Theory

Robert Harper and Daniel R. Licata

1 Overview

Intuitionistic type theory [43] is an expressive formalism that unifies mathematics and computation. A central concept is the *propositions-as-types* principle, according to which propositions are interpreted as types, and proofs of a proposition are interpreted as programs of the associated type. Mathematical propositions are thereby to be understood as specifications, or problem descriptions, that are solved by providing a program that meets the specification. Conversely, a program can, by the same token, be understood as a proof of its type viewed as a proposition. Over the last quarter-century type theory has emerged as the central organizing principle of programming language research, through the identification of the informal concept of language features with type structure. Numerous benefits accrue from the identification of proofs and programs in type theory. First, it provides the foundation for integrating types and verification, the two most successful formal methods used to ensure the correctness of software. Second, it provides a language for the mechanization of mathematics in which proof checking is equivalent to type checking, and proof search is equivalent to writing a program to meet a specification.

Recent research by several authors [9, 25, 28, 34, 41, 64, 65, 69] has exposed a surprising and deep correspondence between identity types in type theory, higher-dimensional morphisms in category theory, and algebraic structures that arise in homotopy theory. This correspondence suggests a fundamental generalization of type theory to account for *higher-dimensional structure*. This structure arises from a careful analysis of equality of elements and types in general, and in particular the behavior of type-indexed families of types (dependent types) under such identifications.[1] The resulting *higher-dimensional type theory* will have significant applications, both to mathematics and to programming. On the mathematical side, Voevodsky [65] is proposing to use type theory as a comprehensive foundation for mechanizing advanced mathematics, as it coheres with contemporary practices much better than well-known formalisms such as axiomatic set theory.

In the work proposed here, we will investigate the benefits of higher-dimensional type theory for programming. We will study the **foundations** of higher-dimensional type theory, by designing a syntactic type theory that supports higher-dimensional concepts and investigating its semantics in category theory and homotopy theory. We will also develop **applications** of higher-dimensional type theory to software development and verification. In particular, we argue that higher-dimensional type theory offers benefits for modular programming with dependent types, generic programming via universes, and software verification using domain-specific specification logics.

Higher-Dimensional Type Theory

To explain the concept of higher-dimensional structure in types, we begin by reviewing the central concept of dependent type theory, that of a *type-indexed family of types*, or *family* for short. A family $\{ B(x) \}_{x \in A}$, assigns a type $B(M)$ to each element $M \in A$. Moreover, the assignment must respect equality of indices in the sense that if M and N are equal elements of A, then $B(M)$ and $B(N)$ are equal types. In intensional type theory [32, 53], this principle holds for *definitional equality*, which expresses purely *computational*

[1] To avoid possible confusions, it is important to stress that *dimensionality*, the focus of this proposal, relates only weakly to the concept of a *universe level* characteristic of predicative type theories. While objects of higher universe level tend to also be higher-dimensional (in a sense to be detailed below), higher-dimensional types exist and are important even at the lowest level of the universe hierarchy.

equivalences such as calculating the sum of two natural numbers. Equal types classify and equate the same elements as one another, so that whenever M and N are definitionally equal elements of A, we have $P \in B(M)$ iff $Q \in B(N)$, and moreover $P \equiv Q \in B(M)$ iff $P \equiv Q \in B(N)$. Definitional equality is, therefore, an *analytic* judgement, one that is justified by direct calculation, without the need for proof [44].

Mathematical practice, however, demands consideration of coarser notions of *equivalence* whose justification may require proof. For example, σ_1 and σ_2 may be computationally distinct sequences of natural numbers (functions of type $N \to N$), yet be regarded as equivalent if for every $n \in N$, $\sigma_1(n) = \sigma_2(n) \in N$. In general such an equation requires justification in the form of an inductive proof that, in effect, considers each $n \in N$ in turn. Similarly, it is often useful in practice to identify isomorphic types [20] such as $A \times B$ and $B \times A$. Such equations are *synthetic* judgements, those that require evidence in the form of a proof. For example, to show that two types are isomorphic requires that we exhibit a pair of mutually inverse functions between them: isomorphism is a *structure*, not a pure *property*.

The *identity type* [53] of intensional type theory, $\mathsf{Id}_A(a, b)$, classifies proofs of "equality" of elements $a, b \in A$. Surprisingly, the rules for the identity type are sound not only for equality, but for these richer notions equivalence. This was first observed by Hofmann and Streicher [34], who gave an interpretation of type theory in which Id_A is interpreted using an "equivalence relation with evidence", or a *groupoid* [34]—a category in which every map is invertible. To emphasize this more general interpretation, we write $a \simeq b \in A$ for $\mathsf{Id}_A(a, b)$. Reflexivity is witnessed by $\mathsf{refl} : a \simeq a \in A$, symmetry by the inverse $\alpha^{-1} : b \simeq a \in A$ of $\alpha : a \simeq b \in A$, and transitivity by composition $\beta \circ \alpha : a \simeq c \in A$ of $\alpha : a \simeq b \in A$ and $\beta : b \simeq c \in A$. To form a groupoid, these forms of evidence must satisfy algebraic laws specifying that composition is associative with reflexivity as identity element, and that the composition (on either side) of a piece of evidence with its inverse yields reflexivity (the identity).

The elimination form for the identity type states that families of types must respect equivalence of indices: if B is an A-indexed family of types, then $\alpha : a \simeq b \in A$ induces a mapping from $B(a)$ to $B(b)$ (and, by symmetry, *vice versa*). Computationally, this action on proofs describes how to lift an equivalence at A to an equivalence at B. Thus, each family $\{ B(x) \}_{x \in A}$ is equipped with an action both on elements $a \in A$ and on proofs $\alpha : a \simeq b \in A$, so that $B(\alpha) : B(a) \simeq B(b)$ is evidence of equivalence of instances of the family. Bearing in mind that groupoids are categories, this says that B forms a *functor* on the groupoid of proofs of equivalence of elements of A.

Moreover, identity types can describe structures more general than a groupoid, because the equations relating refl, $\circ$, and $-^{-1}$ are required to hold only up to higher equivalences—that is, it may require further evidence, such as $\delta : \alpha \circ \alpha^{-1} \simeq \mathsf{refl} : a \simeq a \in A$, to establish these algebraic laws. This leads to the interpretation of types as *weak ω-groupoids* [41, 64], in which equivalence of elements of a type exhibits a groupoid structure whose equational properties hold only up to a higher notion of equivalence. Adapting the terminology of n-categories, we say that the elements of a type are 0-cells, that evidence for equivalence of elements are 1-cells, evidence for equivalence of evidences for equivalence are 2-cells, and so on without end. This *higher-dimensional structure* of types is the subject of intense interest in both the context of this proposal and for the mechanization of advanced mathematics [9, 65]. In particular the concept of a weak ω-groupoid plays a central role in abstract accounts of homotopy theory [9, 65, 69].

In many situations, however, it is sufficient to consider only low-dimensional types, those for which the higher cells are trivial, in that any two proofs of equivalence are themselves equivalent. We say that a type is d-dimensional if all d'-cells are trivial in this sense for $d' > d$. A type in a conventional dependent type theory with uniqueness of identity proofs[2] is, in this terminology, 0-dimensional, meaning that any two proofs of equivalence are themselves equivalent. We call such types *sets* to emphasize that they have no interesting structure besides their elements. Under the principle of proof irrelevance, individual propositions are -1-dimensional because any two proofs of a proposition are identified. The collection of n-dimensional

[2] any two terms of type $Id_A(a, b)$ are themselves propositionally equal

types naturally forms a $n+1$-dimensional type. For example, the type Prop forms a set of propositions considered modulo interprovability. More interestingly, a type Set of sets considered modulo *isomorphism* is 1-dimensional—because there can be many different isomorphisms between two sets. This is a natural first example of a higher-dimensional type, which goes beyond traditional type theories by having a computationally relevant notion of equivalence. Because a traditional type theory has only sets (0-dimensional types), we say that the type theory as a whole is 1-dimensional. A dependent type theory in which types themselves may be 1-dimensional, like Set, is called *2-dimensional dependent type theory*, or $2TT$; this theory forms the focus of this proposal.

Identity types in intensional type theory are compatible with richer notions of equivalence, but current type theories do not expose this fact to the programmer: there is no way to define higher-dimensional types, nor to exploit the functorial action of the type constructors. Voevodsky's univalence axiom [65] is one way to remedy this, but it does not provide the computational behavior of functoriality definitionally. One contribution of the proposed work is to investigate a new syntactic type theory that accounts for these higher-dimensional concepts.

Applications. Higher-dimensional type theory will have significant applications to the formalization of mathematics, because it is common mathematical practice to treat equivalent objects as interchangeable. Here, we propose to investigate analogous applications of this idea in programming. In computational terms, higher-dimensional type theory equips each type and term with an action on equivalences, which can be thought of as a *generic program* that is defined for all types in the language. This is related to previous work on generic traversals of data structures, which automatically lift a transformation on a type to data structures that contain occurrences of that type [11, 36]. For example, if $d \in N \to N$ is the doubling function on natural numbers, then d may be lifted to types such as $N \times N \to N \times N$ by applying d to either, or both, components of a pair. The choice of which, and the code to do it, is determined by specifying which of the two occurrences of N in the type $N \times N$ is to be the locus of activity. The desired transformation arises as the *functorial action* of the corresponding type constructor. For example, $d \times N$ doubles the first component, $N \times d$ the second, and $d \times d$ both components. These are, respectively, the actions of the type constructors $(-) \times N$, $N \times (-)$, and $(-) \times (-)$.

In higher-dimensional type theory, we write such generic programs by choosing an appropriate notion of equivalence for a type A, whereupon the type theory ensures that equivalence at A can be lifted to any family $\{ B(x) \}_{x \in A}$. For example, consider a type Set whose elements are sets and whose equivalences are isomorphisms between them, so that $A \times B$ and $B \times A$ are regarded as equivalent sets. We may lift this postulated isomorphism to more complex data structures, simply by specifying an operator on sets, and applying its functorial action. If $F : \mathsf{Set} \to \mathsf{Set}$ is a set-indexed family of sets, then the principle of respect for equivalence dictates that $F(A \times B)$ is equivalent to $F(B \times A)$, which means, as sets, they must be isomorphic. The required isomorphism is simply the functorial action of F on the isomorphism $i : A \times B \cong B \times A$, yielding $F(i) : F(A \times B) \cong F(B \times A)$. The program $F(i)$ is determined generically from the isomorphism i.

This idea solves numerous problems with current dependently typed programming practice. The first is **modularity** via type abstraction: To promote code reuse and maintainability, it is good practice to hide the implementation of types and programs from clients. However, in a dependently typed language, this curtails opportunities for after-the-fact reasoning about these programs. A solution is to export a *view* [45, 47, 66] of an abstract type t as a concrete type T, and of each operation on t as a corresponding operation on T. Then clients may code and reason using the view, while implementers are free to exploit any representation that supports the same view. ωTT supports this style of programming: the view is represented as an equivalence $t \simeq T \in \mathsf{type}$ between the abstract and concrete types, and corresponding equivalences between operations. The generic lifting of an isomorphism is crucial for stating and programming with these equivalences, as we

illustrate below.

The second is **code reuse** in the presence of datatype-generic programming via universes [4, 13]. A universe circumscribes a particular collection of types on which certain functions can be defined once-and-for-all, saving programmers the effort of implementing the operation for the datatypes specific to their programs. For example, it is possible to define universes that support a generic equality test, generic parsing and serialization, generic traversals, and generic zippers (cursor data structures that facilitate algorithms that zoom in on a part of a structure, modify it, and then put the result back together) [1, 4, 5]. Universes can also be used to embed domain-specific type systems, such as modal types for distributed computing [39] and contextual types for programming with abstract syntax [38]. Formally, a universe consists of a type U of *codes* for types, along with a function $El(-) : U \to \mathsf{type}$ that maps each code to the type it represents. Generic programs are written by recursion on codes. A problem with this style of programming is that the proliferation of universes with different generic operations creates multiple isomorphic copies of each data type: lists that can be parsed and printed are represented differently than traversable lists, or lists with zippers, and so on. ωTT's generic support for isomorphisms will automatically allow code written for one universe to be used with code written for another.

The third is **reasoning** about polymorphic programs. When verifying programs that use a polymorphic operation $f : \forall\alpha : \mathsf{Set}.A(\alpha) \to B(\alpha)$ it is often necessary to use the fact that different instances of f are related by *naturality*. This kind of reasoning has been studied in external parametricity logics and relational interpretations [14, 57] and in previous category-theoretic accounts of polymorphic (but not dependently typed) languages [10]. In ωTT, these reasoning principles are available inside the type theory, using the action on equivalences of such a term f: if $\alpha : a \cong b \in Set$, then $f_b = B(\alpha) \circ f_a \circ A(\alpha^{-1})$. Thus, various "free theorems" [67] are available as reasoning principles.

Directed Type Theory

Thus far, we have considered only symmetric higher-dimensional structures. Symmetry of equivalence plays an important role in the functorial action of type constructors. For example, suppose that $F(-)$ sends $X \in \mathsf{Set}$ to $X \to X \in \mathsf{Set}$. The action of F on $i : A \cong B$ is the function $i^{-1} \to i \in (A \to A) \to (B \to B)$ that transforms $b \in B$ to $i(f(i^{-1}(b)))$, where $f \in A \to A$. Symmetry of equality, which is expressed by the existence of inverse maps, is critical to ensuring that every set constructor, $F \in \mathsf{Set} \to \mathsf{Set}$, have an action on proofs of equality (that is, isomorphisms) of sets. Without this assumption, the required action need not exist. This observation is critically important in the context of this proposal, as a second major contribution of the proposed work will be to show that *the assumption of symmetry can be relaxed*.

We propose to study a *higher-dimensional directed dependent type theory* (ωDTT), which is based on relaxing the groupoid interpretation of equivalence (which demands symmetry by requiring inverses) to a *categorial* interpretation (which allows, but does not require, that every map have an inverse). There are several motivations for this generalization. First, the development of *directed homotopy theory* [29] demands consideration of non-symmetric transformations between paths in a space. Second, and more importantly for present purposes, there are good applications to programming of *directed* transformations between the instances of a type family determined by a directed relationship between the instantiating indices. Returning to the applications discussed above: For modularity, directed types will support one-way views, which decouple the methods for constructing and deconstructing an abstract type. For generic programming, directed types will allow generic use not just of type isomorphisms, but general *coercions* between types, as arise, for example, in subtyping. For reasoning, directed types will provide more general forms of naturality, such as the familiar principle that a polymorphic function commutes with the map function on lists. We also propose to investigate two additional applications: First, directedness facilitates the implementation of domain-specific logics and their use in verifying software, as we discuss in detail below. Second, directedness may have applications to formalizing the metatheory of programming languages, which are rife with

directed phenomena—e.g., reduction in operational semantics and monotone functions in domain-theoretic denotational semantics, both of which have been analyzed in categorical terms [60, 61, 68].

The natural semantic setting for ωDTT is that of *weak ω-categories*, in which the characteristic axioms hold for an n-cell only up to equivalences given by symmetric $n+1$-cells (but not all higher-dimensional cells need be equivalences). The restriction to the 2-dimensional case, called 2DTT, amounts to associating, with each type, a category of transformations between elements (generalizing a pre-order with evidence) such that families of types respect these transformations. That is, writing $\alpha : a \lesssim b \in A$ for a transformation of a into b as elements of A, if B is an A-indexed family of types, then we ask that B determine a transformation $B(\alpha) : B(a) \lesssim B(b)$ on instances of the family. That is, the family should determine a functorial action on the transformations between elements of the type.

It is immediate that directed type theory generalizes higher-dimensional symmetric type theory, in that every groupoid is *a fortiori* a category. But it is less obvious how to formulate 2- (or higher-) dimensional directed type theory. We propose to investigate the following **foundational** issues, which arise immediately in an attempt to formulate 2DTT:

1. The transformations $\alpha : a \lesssim b \in A$ may no longer be represented as elements of the Martin-Löf identity type, for the simple reason that the identity type is inherently symmetric. (That is, symmetry of equality is derivable using the elimination form for the identity type.) Instead, we treat $\alpha : a \lesssim b \in A$ as a basic judgement form, rather than as an instance of type membership $\alpha \in \mathsf{Id}_A(a, b)$.

2. In the absence of symmetry not all type families admit functorial actions; the *variances* of the family must be considered [2, 16, 23, 62]. For example, the set operator $(-) \to N$ is *contravariant*, the operator $N \to (-)$ is *covariant*, and the operator $(-) \to (-)$ is *non-variant*. Directed type theory must take account of variances from the outset, in sharp contrast to the symmetric case.

3. The analog of symmetric type theory's quotient types is the notion of an *internal category* in directed type theory, which will allow programmers to define their own directed types by giving a collection of elements and a notion of transformation between them.

4. The concept of the *opposite* of a category, which ordinarily plays a central role in accounting for variances, is much more subtle in the dependent (fibered) case than in the non-dependent case usually considered. A full accounting of variances involves a modality for contravariance that demands further investigation.

5. Semantically, it is clear that one should be able to internalize the judgement $\alpha : a \lesssim b \in A$ by a type $\mathsf{Hom}_A(a, b)$ of transformations between $a, b \in A$, as long as a is treated as a contravariant position. However, the introduction and elimination rules for this type require further study.

6. It is paramount to adapt familiar type constructors, such as an inductive datatype mechanism, to the directed case. Some type constructors interact with variances in interesting ways. For example, there are two forms of dependent function type, one that is contravariant in its domain (as might be expected), and one that is covariant in its domain (which, surprisingly, exists, and is important to the main application of 2DTT described below).

We propose to investigate these issues, and others as may arise, in the formulation of a syntactic higher-dimensional type theory. In addition, we propose to investigate its semantics in higher-dimensional category theory and homotopy theory.

Applications. In addition to the applications discussed above, we propose to develop the practical application of 2DTT to **programming with domain-specific specification logics**. Our previous project on

Integrating Types and Verification shows that the use of logics specific to an application domain is a promising way to verify software within a dependently typed programming language. Examples of domain-specific logics include separation logic [59], which has been used to verify imperative programs [51], and authorization logics, which have been used to verify security properties in security-typed languages [8, 35, 46, 63], as investigated in our previous project on *Manifest Security*. Our initial investigation into 2DTT arose out of a desire to design a better meta-language for defining specification logics such as these.

The central notion in logic is consequence—entailment from premises to conclusions—and, in our previous project, we identified two notions of consequence necessary for programming with logics: *derivability*, which captures uniform reasoning, and *admissibility*, which captures inductive proofs and functional programs. Derivability is necessary for representing the syntax of propositions and proofs, whereas admissibility is necessary to prove theorems and implement theorem provers. Presently, derivability is better supported in LF-based proof assistants, such as Twelf, Delphin, and Beluga [30, 55, 56, 58], whereas admissibility is better supported in proof assistants based on Martin-Löf type theory, such as Coq, Agda, and Epigram [19, 43, 45, 54]. In our previous project, we began to explore an approach to reconciling these differences by offering better support for derivability inside of Martin-Löf type theory. Our approach uses *pronominal representations* of syntax in the style originally advocated by de Bruijn [21, 52] in which variables are regarded as *pronouns* that refer to a binding site that fixes their referents. Semantically, pronominal representations can be analyzed using the category theoretic concepts of functors and monads [3, 24, 33].

2DTT will allow us to put this theory directly into practice, and to generalize it in important ways: First, by giving a simple description of the syntax of the propositions and proofs of a specification logic in 2DTT, programmers will have access to a generic implementation of the *structural properties* of consequence relations. These structural properties are essential for programming with domain-specific logics, and the burden of implementing them manually for each logic reduces the applicability of this verification technique. In 2DTT, the structural properties arise naturally out of the language-wide notion of respect for transformations. Second, 2DTT affords a language of signatures that significantly extends that of logical frameworks such as LF, because representations of logics may exploit admissibility (functional programs), an idea we explored in the simply-typed case in our previous project [38, 40]. This enables a wide class of more convenient representations of logics, encompassing negated premises in inference rules, infinitary sequent calculi for inductive types [31], and domain-specific languages and logics that inherit pattern-matching from the host language [71].

To demonstrate these applications, it is essential to develop an implementation of 2DTT. While a full-scale implementation comparable to current systems such as Coq [19] or Agda [54] is beyond the scope of the present proposal, we propose to develop a prototype as a proof of concept and as a foundation for future work on the implementation of higher-dimensional directed dependent type theory. Specifically, we will develop a type checker for 2DTT that is capable of verifying the static correctness of programs written in the language. This core type checker will serve as a foundation for an elaboration process that translates a more user-friendly language into the fundamental type theory. Elaboration would be responsible for such tasks as argument synthesis (a generalization of type inference) and variance inference.

Summary

We propose to investigate a fundamental generalization of type theory to account for higher-dimensional and directed structures. This generalization is based on a semantic correspondence with higher-dimensional category theory and homotopy theory, which is ripe for exploitation. The design of a higher-dimensional and directed type theory requires foundational work on reflecting these semantic structures in a programming language. In return, the resulting theory will have significant applications, both to dependently typed programming, by improving modularity and code reuse, and to software verification, by facilitating the construction of domain-specific logics and providing powerful reasoning principles for polymorphic code. In

$$\textit{Contexts:}\quad \frac{\Gamma\ \mathsf{ctx}}{\Gamma^{\mathsf{op}}\ \mathsf{ctx}} \qquad \frac{\Gamma\ \mathsf{ctx} \quad \Gamma \vdash A\ \mathsf{type}}{\Gamma\,,\, x{:}A^{+}\ \mathsf{ctx}} \qquad \frac{\Gamma\ \mathsf{ctx} \quad \Gamma^{\mathsf{op}} \vdash A\ \mathsf{type}}{\Gamma\,,\, x{:}A^{-}\ \mathsf{ctx}}$$

$$\textit{Functoriality:}\quad \frac{\Gamma, x{:}A^{+} \vdash B\ \mathsf{type} \quad \Gamma \vdash \alpha : M_1 \lesssim_A M_2 \quad \Gamma \vdash M : B[M_1/x]}{\Gamma \vdash \mathsf{map}_{x:A^{+}.B}\ \alpha\ M : B[M_2/x]}$$

$$\textit{Function types:}\quad \frac{\Gamma^{\mathsf{op}} \vdash A\ \mathsf{type} \quad \Gamma\,,\, x{:}A^{-} \vdash B\ \mathsf{type}}{\Gamma \vdash \Pi\, x{:}A.\, B\ \mathsf{type}} \qquad \frac{\Gamma\,,\, x{:}A^{-} \vdash M : B}{\Gamma \vdash \lambda\, x.\, M : \Pi\, x{:}A.\, B} \qquad \frac{\Gamma \vdash M_1 : \Pi\, x{:}A.\, B \quad \Gamma^{\mathsf{op}} \vdash M_2 : A}{\Gamma \vdash M_1\ M_2 : B[M_2/x]}$$

$$\frac{\Gamma\,,\, x{:}A^{-} \vdash \alpha : (M\ x) \lesssim_B (N\ x)}{\Gamma \vdash \lambda\, x.\, \alpha : M \lesssim_{\Pi\, x:A.\, B} N} \qquad \frac{\Gamma \vdash \alpha : M \lesssim_{\Pi\, x:A.\, B} N \quad \Gamma^{\mathsf{op}} \vdash \beta : N_1 \lesssim_A M_1}{\Gamma \vdash \alpha\, \beta : \mathsf{map}_B\ \beta\ (MM_1) \lesssim_{B[N_1/x]} (NN_1)}$$

Figure 1: 2DTT: Key Constructs

the remainder of this proposal, we describe the 2DTT type theory in more technical detail, and then sketch the extensions and applications of 2DTT that constitute the proposed work.

2 Technical Details

2.1 2DTT

In this section, we outline a preliminary account of 2-dimensional directed type theory, 2DTT [37]. Our goal is to illustrate the two main technical differences between directed type theory and ordinary symmetric type theory: First, the specification of a typical dependent type theory consists of three main forms of syntax, contexts Γ ctx, dependent types $\Gamma \vdash A$ type, and terms $\Gamma \vdash M : A$. Because transformations cannot be represented by the identity type, 2DTT adds a fourth form of syntax, transformations $\Gamma \vdash \alpha : M \lesssim_A N$. In addition to type-generic rules giving reflexivity and transitivity, there are specific transformation rules for each type A. For example, a transformation between natural numbers $m \lesssim_{\mathsf{nat}} n$ is a proof of equality of m and n. A transformation between functions $f \lesssim_{A \to B} g$ is a point-wise transformation that for every $x : A$ gives a transformation $f\ x \lesssim_B g\ x$. Next, consider a universe set of 0-dimensional types, equipped with a function $El(S)$ that maps each set to a type. A transformation between sets $S_1 \lesssim_{\mathsf{set}} S_2$ is a function $El(S_1) \to El(S_2)$ from elements of S_1 to elements of S_2.

Additionally, 2DTT includes a construct map which states that types respect transformation: For any type $x : A \vdash C$ type, a transformation $\alpha : M_1 \lesssim_A M_2$ induces a function $\mathsf{map}_{x:A.C}\ \alpha$ from $C[M_1]$ to $C[M_2]$. For natural numbers, map says that equal natural numbers determine transformable types—in this case the transformation is the identity. For sets, map says that any type $x : \mathsf{set} \vdash C$ type with a free set variable induces a higher-order function $(El(S_1) \to El(S_2)) \to C[S_1] \to C[S_2]$. For example, when $C = \mathsf{list}(x)$, $\mathsf{map}_{x.\mathsf{list}(x)}$ is the standard map function on lists, which applies the given function to each element. In this case, map is not the identity—it has real computational content.

Second, to describe all of the usual type constructors, 2DTT must track the *variance* of type constructors, because $\to$ is *contravariant* in the domain, but covariant in the range. We accomplish this using two technical devices: First, each assumption in the context is annotated with a variance, which is either $^{+}$ or $^{-}$. Second, there is a *dualization* operation on contexts, written Γ^{op}, which swaps the variance of each assumption.

This is made precise in the rules in Figure 1. The first line of rules define contexts: If Γ is a context, then its dual Γ^{op} is also a context. A context can be extended by a *covariant* assumption $x : A^{+}$, in which case the

type A must make sense in Γ, and additionally by a contravariant assumption, in which case A must make sense in Γ^{op}.

The next rule describes map: given a type B with free variable $x : A^+$, a transformation α induces a function from $B[M_1]$ to $B[M_2]$. In fact, a more general version of map is possible: the map rule in the figure allows transformation at one covariant assumption $x : A^+$, but in general we may allow simultaneous transformation of any number of co- and contravariant assumptions.

The next set of rules describe dependent functions, which illustrate the methodology for defining types in 2DTT. The type $\Pi\, x{:}A.\, B$ is well-formed if A is well-formed *contravariantly* in Γ, and B is well-formed covariantly in Γ, with x as a new contravariant assumption. The terms for dependent functions are the standard λ and application forms, adjusted to account for variance: When typing $\lambda\, x{:}A.\, M$, A is assumed contravariantly; when typing $M_1\ M_2$, the argument M_2 is typed contravariantly. Using these rules, we can give the following computation rule for map at function type:

$$\mathsf{map}_{\Pi\, x:A.\, B}\ \alpha\ f \equiv \lambda\, x.\, \mathsf{map}_B\ \alpha\ (f\ (\mathsf{map}_A\ \alpha\ x))$$

This rule says that a function f is transformed by pre-composing with transformation in its domain type A, and post-composing with transformation in its range type B. The contravariance of A is necessary for $(\mathsf{map}_A\ \alpha\ x)$ to be oriented in the correct direction.

The next two rules define transformations at Π-types: a transformation is introduced by giving a pointwise transformation, as discussed above, and eliminated by the principle that transformable functions, applied to any transformable arguments, are themselves transformable. The direction of the transformation between arguments is reversed, which is appropriate for contravariant positions.

2DTT has a straightforward semantics in Cat, the 2-category of categories, functors, and natural transformations [37]; investigating additional models is an interesting piece of our proposed work.

2.2 Universes

A natural example of a higher-dimensional type is a universe, a type whose members themselves represent types. For example, we may define a universe setwhose members represent 0-dimensional types, and whose transformations are functions:

$$\frac{}{\Gamma \vdash \mathsf{set}\ \mathsf{type}} \quad \frac{\Gamma \vdash S : \mathsf{set}}{\Gamma \vdash \mathit{El}(S)\ \mathsf{type}} \quad \frac{\Gamma\,, x{:}\mathit{El}(S)^+ \vdash M : \mathit{El}(S')}{\Gamma \vdash x.M : S \lesssim_{\mathsf{set}} S'} \quad \frac{}{\Gamma \vdash \star : M \lesssim_{\mathit{El}(S)} M} \quad \frac{\Gamma \vdash \alpha : M \lesssim_{\mathit{El}(S)} N}{\Gamma \vdash M \equiv N : \mathit{El}(S)}$$

The first rule says that set is a type, and the second that any term of type set determines a type $\mathit{El}(S)$ classifying the members of S. The next rule says that a transformation between sets S and S' consists of a function $x.M$ from $\mathit{El}(S)$ to $\mathit{El}(S')$. On the other hand, because sets are 0-dimensional, the only transformation at $\mathit{El}(S)$ is the identity, as the next two rules state—we write $\equiv$ for the *equality* judgement on terms, which is a finer notion than equivalence.[3] In this case, transformation at $\mathit{El}(S)$ is symmetric, and we sometimes write $M \simeq_{\mathit{El}(S)} N$ to emphasize this point.

Sets are closed under the usual dependent type constructors, such as dependent functions $\Pi\, x{:}S.\, S'$ and pairs $\Sigma\, x{:}S.\, S'$, as well as empty (0), unit (1), boolean (2), and inductive and coinductive types (nat, $\mathsf{list}(S)$, ...). We elide the rules defining these sets and their members; the type constructors take account of variance in the same way as above (e.g. in $\Pi\, x{:}S.\, S'$, S is a contravariant position). Each set determines an action on transformations; for example $\mathsf{map}_{a\,:\,\mathsf{set}^+.\mathit{El}(\mathsf{list}(a))}$ is the familiar map function on lists. It is a common notational convenience to elide the coercion from sets to types, writing S for $\mathit{El}(S)$; we follow this convention below.

[3] These transformation rules treat transformation at $\mathit{El}(S)$ like equality in extensional type theory [18].

key	:	$\mathsf{set}^{\simeq}$	show	:	$\mathsf{dict} \lesssim_{\mathsf{set}} \mathsf{list}(\mathsf{key} \times \mathsf{val})$
value	:	$\mathsf{set}^{\simeq}$	hide	:	$\mathsf{list}(\mathsf{key} \times \mathsf{val}) \lesssim_{\mathsf{set}} \mathsf{dict}$
dict	:	$\mathsf{set}^{\simeq}$	retract	:	$(\mathsf{show} \circ \mathsf{hide}) \simeq \mathsf{refl}$
lookup	:	$\mathsf{dict} \to \mathsf{key} \to \mathsf{value\ option}$	lkspec	:	$\mathsf{lookup} \simeq \mathsf{map}_{d:\mathsf{set}^-.d \to \mathsf{key} \to \mathsf{value\ option}}\ \mathsf{show}\ \mathsf{L.lookup}$
insert	:	$\mathsf{dict} \to \mathsf{key} \to \mathsf{value} \to \mathsf{dict}$	insspec	:	$\mathsf{insert} \simeq \mathsf{map}_{d:\mathsf{set}^+,d':\mathsf{set}^-.d' \to \mathsf{key} \to \mathsf{value} \to d}\ (\mathsf{hide}/d, \mathsf{show}/d')\ \mathsf{L.insert}$

Figure 2: View for a Dictionary ADT in 2DTT

An alternative is to define a symmetric universe of sets $\mathsf{set}^{\simeq}$, where transformation is isomorphism:

$$\frac{\Gamma\,,\, x{:}El(S)^{+} \vdash f : El(S') \quad \Gamma\,,\, y{:}El(S')^{+} \vdash g : El(S) \quad f[g/x] \equiv y \quad g[f/y] \equiv x}{\Gamma \vdash : S \lesssim_{\mathsf{set}^{\simeq}} S'}$$

The advantage of the symmetric universe is that all constructions possible in standard dependent type theory are possible; in particular, the rules for $\mathsf{set}^{\simeq}$ constructors need not account for variances. The disadvantage of the symmetric universe is that map can only be used to lift isomorphisms, not arbitrary functions, which provides a less useful generic program.

As we illustrate below, both directed and symmetric universes are useful, and studying the relationships between them is an important component of the proposed work. For example, there are both co- and contravariant inclusions from $\mathsf{set}^{\simeq}$ to set that choose the appropriate side of the isomorphism.

2.3 Application: Modularity and Code Reuse

We are now in a position to sketch an illustrative example of using 2DTT to specify an abstract data type. We implement a dictionary data structure, mapping keys to values, where the implementation of the dictionary type, and the operations on it, are held abstract. However, the interface provides a view of the dictionary in terms of a simple but inefficient implementation using association lists. This view can be used both for programming—converting a dictionary to and from a list—and, more importantly, for after-the-fact verification, from the "outside" of the abstract type.

We show the dictionary interface in Figure 2. First, we define abstract types key, value, and dict, represented as assumptions of type $\mathsf{set}^{\simeq}$. As discussed above, these assumptions can be freely used in positions of either variance, and we exploit this fact in defining various operations on these abstract types, in this case lookup and insert.

Next, we define the view: transformations show and hide that relate dict to $\mathsf{list}(\mathsf{key} \times \mathsf{val})$. The intention is that dict is implemented by some efficient representation, such as a balanced binary tree, with the list view representing the dictionary in extension, as a list of ordered pairs. Thus, we require the composition $\mathsf{show} \circ \mathsf{hide}$ to be the identity: retract states that showing a dictionary determined by its extension should give back that extension. We do not require the converse, $\mathsf{hide} \circ \mathsf{show}$, to be the identity, because there may be many internal representations with the same extension—e.g. different arrangements of the tree. Because this pair does not constitute an isomorphism, this example is not programmable using the identity type in symmetric higher-dimensional type theory.

On the implementation side, show and hide are written by giving functions $\mathsf{dict} \to \mathsf{list}(\mathsf{key} \times \mathsf{val})$ and vice versa. On the client side, these transformations can be used to coerce a dictionary to a list and back using map. An example of this is in specifying the behavior of lookup and insert: lkspec and insspec state that lookup and insert behave like the coercion of the equivalent operations on the list implementation by hide and show. This coercion is represented using map, using an annotation that identifies the appropriate positions in the type to coerce. For lookup, this is the argument dictionary, which is a contravariant position; for insert, this is the argument dictionary (represented by the contravariant variable d') and the result

(represented by the covariant variable d). The calls to map supply an appropriate transformation for each position: using the computation rules for map, these assumptions expand to

$$\begin{array}{l}\mathsf{lookup} \simeq \lambda\, d, k.\, \mathsf{L.lookup}\ (\mathsf{map\ show}\ d)\ k \\ \mathsf{insert} \simeq \lambda\, d, k, v.\, \mathsf{map\ hide}\ (\mathsf{L.insert}\ (\mathsf{map\ hide}\ d)\ k\ v)\end{array}$$

which say that the abstract operations behave like the specification operations.

This interface can be used to derive properties of the abstract implementation using its specification. For example, we can prove

$$\mathsf{lookup}\ (\mathsf{insert}\ d\ k\ v)\ k \simeq \mathsf{some}(v)$$

because, after expanding the definition of map, and collapsing show$\circ$hide using retract, the equation reduces to the corresponding property of the specification.

The benefit of 2DTT for this style of programming, relative to current type theories, is that views can be automatically lifted to the types of the operations using map. In fact, the example can be made even more concise by packaging up the operations on the abstract type:

$$\begin{array}{lll}\mathsf{dictmethods}(\mathsf{dict}^+, \mathsf{dict}^-) \equiv \{ & \mathsf{lookup} : \mathsf{dict}^- \to \mathsf{key} \to \mathsf{value\ option} \\ & \mathsf{insert} : \mathsf{dict}^- \to \mathsf{key} \to \mathsf{value} \to \mathsf{dict}^+\}\end{array}$$

This specifies the operations in terms of a covariant type dict^+ and a contravariant type dict^-. Then lookup, insert, lookupspec, and insertspec can be replaced by the following:

$$\begin{array}{lcl}\mathsf{m} & : & \mathsf{dictmethods}(\mathsf{dict}, \mathsf{dict}) \\ \mathsf{mspec} & : & \mathsf{m} \simeq \mathsf{map}_{d:\mathsf{set},d':\mathsf{set}^-.\mathsf{dictmethods}(d,d')}\ (\mathsf{hide}/d, \mathsf{show}/d')\ \mathsf{L.m}\end{array}$$

While the increased conciseness is modest in this small example, the ability to automatically derive these specifications will be more important with interfaces with tens or hundreds of operations, where writing and maintaining them would be impractical. Additionally, 2DTT provides type-generic reasoning principles, which can be exploited in proofs. Moreover, here, we have verified a simply-typed program using dependent types, but the approach scales to dependently typed programs as well, as we discuss below.

The application of 2DTT to code reuse has a similar flavor: map can be used to lift an isomorphism to an interface of operations, mediating between the different but isomorphic representations of types that arise when doing generic programming.

2.4 Application: Domain-Specific Logics

In our previous project on *Integrating Types and Verification*, we studied the use of domain-specific logics to verify software. Examples of domain-specific logics include separation logic [59], which has been used to verify imperative programs in Ynot [51], and authorization logics, which have been used to verify security properties in security-typed languages [8, 35, 46, 63].

For example, in Ynot, imperative programs are specified using Hoare Triple Types $\{P\}A\{Q\}$, which classify a command that, if precondition P holds, returns a value of type A, in a state satisfying Q. Here P and Q are predicates on the heap. For example:

$$\mathsf{write} : \forall l, v.\{\exists v'.(l \mapsto v')(\mathit{before})\}\mathsf{unit}\{\mathit{after} = \mathit{before}[l \mapsto v]\}$$

This specification for writing a value v to a memory cell l says that if the location l points to some value v' in the initial heap (*before*), then the final heap (*after*) is the initial heap updated with l pointing to the written value v. Ynot has been used to verify various imperative data structures [17], such as stacks, queues, hash

tables, binary search trees, and binomial trees; algorithms, such as parsing [17]; and libraries, such as a web services interface [70] and database management software [42].

Security typed programming languages use authorization logics to specify decentralized access control policies, where access control is expressed as the aggregate of statements by different principals about the resources they control. For example, a clause of a file system policy for a company might say that

$$\forall r.\forall o.\forall f.(\mathsf{HR\ says\ employee}(r) \wedge \mathsf{FS\ says\ own}(o, f) \wedge o\ \mathsf{says\ mayread}(r, f)) \supset \mathsf{mayread}(r, f)$$

That is, if the human resources department says that r is an employee, and the file system says that o owns a file f, then if o says that r mayread f, then r should be granted access to f. This policy refers to statements by several different principals, each of which has authority over different information—who is an employee, who owns a file, who may read a file. Authorization logics have been used to specify file systems [27] and intelligence declassification procedures [26], and security-typed languages have been used to implement medium-sized examples such as a conference management server and an e-mail client [63].

Our initial interest in higher-dimensional type theory arose out of a desire for a better meta-language in which programmers can implement and verify programs with such domain-specific logics. Key activities when programming with domain-specific logics are (1) representing the syntax of propositions and proofs, (2) proving meta-theoretic results about the logic, and (3) implementing theorem provers. A dependently typed programming language can facilitate these activities by offering support for two kinds of functions, derivability and admissibility. 2DTT will support these two concepts better than any existing dependent type theory: it will allow **automatically deriving the *structural properties*** for logics specified by rules that **mix admissibility and derivability**, as we now explain.

Structural Properties The syntax of a domain-specific logic can be represented by a family of types $\psi : \mathsf{ctx}^+ \vdash \mathsf{formula}[\psi]\ \mathsf{type}$ indexed by *contexts* $\psi : \mathsf{ctx}$ consisting of a finite sequence of *parameters*. The values of type $\mathsf{formula}[\psi]$ represent say the formulas of a domain-specific logic generated by a collection of constructors over the parameters in ψ. So, for example, $\mathsf{exists}(v.l \hookrightarrow v)$ ("there exists a value such that location l points to that value) has type $\mathsf{formula}[l]$, since it has one free variable, l. Here $x.t$ is an *abstractor*, which indicates that the parameter, x, is bound within the term t, and hence may be renamed without change of meaning.[4]

A key property of the syntax of formulas is that it is *structural*, which means that it obeys the following properties of variables:

- Reflexivity: $x : \mathsf{formula}[x]$. Parameters are terms.
- Weakening: $\mathsf{formula}[\psi] \lesssim \mathsf{formula}[\psi, \psi']$. Parameters may "occur" vacuously.
- Contraction: $\mathsf{formula}[\psi, x, x, \psi'] \simeq \mathsf{formula}[\psi, x, \psi']$. A parameter may be used more than once.
- Permutation: $\mathsf{formula}[\psi, x, y, \psi'] \simeq \mathsf{formula}[\psi, y, x, \psi']$. The order of parameters is immaterial.
- Substitution: if $t : \mathsf{formula}[\psi, x]$ and $u : \mathsf{formula}[\psi]$, then $[u/x]t : \mathsf{formula}[\psi]$. We may replace a parameter by a term (avoiding capture of parameters bound by abstractors).

The structural properties are essential both for stating inference rules defining the truth of a formula, and for computing with formulas.

In the functorial/monadic approach to abstract syntax [6, 24, 33], weakening, contraction, and permutation are consolidated into the principle of *respect* for the ordering on contexts,

if $\psi \lesssim \psi'$, then $\mathsf{formula}[\psi] \lesssim \mathsf{formula}[\psi']$

[4] Informally, "he" and "she" are equivalent grammatical pronouns when it comes to resolving their referents.

derived from the ordering on contexts given by the following axioms (writing $\psi \simeq \psi'$ for equivalence relation induced by $\psi \lesssim \psi'$):

- $\psi \lesssim \psi, \psi'$.
- $\psi, x, x, \psi' \simeq \psi, x, \psi'$.
- $\psi, x, y, \psi' \simeq \psi, y, x, \psi'$.

The functorial action of $\mathsf{formula}[-]$ implements the weakening, contraction, and permutation transformations on terms. Substitution is accounted for by showing that $\mathsf{formula}[-]$ determines a *relative monad* [7].

Next, we sketch two elements of our proposed work. The first, internal categories, will permit the definition of a one-dimensional type ctx whose terms are contexts and whose transformations give the structural properties. The second will permit the definition of inductive types of formulas $\mathsf{formula}[\psi]$ that are automatically equipped with a generic implementation of the structural properties.

One-dimensional types such as ctx may be defined by specifying an **internal category**—by defining a category inside the type theory:

$$\frac{O : \mathsf{set}^{\simeq} \quad A : O \to O \to \mathsf{set}^{\simeq} \quad r : \Pi\, x{:}O.\, A\ x\ x \quad t : \Pi\, x_1, x_2, x_3{:}O.\, A\ x_2\ x_3 \to A\ x_1\ x_2 \to A\ x_1\ x_3 \quad \ldots}{\mathsf{cat}\{mem = O, \lesssim = A, \mathsf{refl} = r, \circ = t\})\ \mathsf{type}}$$

This says that a programmer may specify a type by giving a set of members of the type (O), a set of transformations between any two members A, equipped with reflexivity and transitivity; the ellipsis elides additional premises stating that reflexivity and transitivity are associative and unital.

For example, a representation of contexts of sorted variables $x_1 : \sigma_1, \ldots x_n : \sigma_n$ in de Bruijn form [21] will be specified by $\mathsf{ctx} = \mathsf{cat}\{mem = \mathsf{list}(\mathsf{sort}), \lesssim = \lambda\, \Psi_1, \Psi_2 . . \Psi_1 \vdash \Psi_2, \mathsf{refl} = \ldots, \circ = \ldots\}$. This definition states that the terms of type ctx are lists of sorts σ, while the transformations are given by a type $\Psi_1 \vdash \Psi_2$ which gives the structural properties of the logic; for example, by choosing $\vdash$ to be variable-for-variable substitutions, we will obtain the structural properties of weakening, exchange, and contraction. $\vdash$ must satisfy identity and composition properties to fill the holes for refl and $\circ$.

Next, syntax can be represented as an **inductive datatype** that is automatically equipped with the structural properties. For example, we may represent syntax using *well-scoped de Bruijn indices* [6, 12, 15] for variables: we will introduce a set $\sigma \in \Psi$ representing proofs that the sort σ is in the list Ψ; the inhabitants are fancily-typed natural numbers, with $\mathsf{0}$ proving $\sigma \in (\Psi, \sigma)$ and s proving $\sigma \in (\Psi, \sigma')$ from $\sigma \in \Psi$. The type $\sigma \in -$ is functorial, with action on an index given by applying the substitution.

We propose to adapt well-known formalisms for inductive datatypes, such as indexed containers [5], to directed type theory. Using such a datatype mechanism, the syntax of a second-order authorization logic

$$\phi ::= \alpha \mid \exists x : \tau.\phi \mid k\ \mathsf{says}\ A \mid \ldots$$

can be described as follows:

$$\begin{array}{lcl} \mathsf{formula} & : & \mathsf{ctx} \to \mathsf{set} \\ \mathsf{formula}\ \psi & \cong & \mathsf{v\ of}\ (\mathsf{formula} \in \psi) \mid \mathsf{exists\ of}\ \Sigma\ s{:}\mathsf{sort}.\ \mathsf{formula}\ (\psi, s) \mid \mathsf{says\ of\ principal}\ \psi \times \mathsf{formula}\ \psi \end{array}$$

The constructor v says that formula variables can be used as formulas; the constructor exists that a formula can be constructed from a sort and a formula in a context extended with that sort; the constructor says states that a formula can be constructed from a principal and a formula.

The advantage of describing this datatype in directed type theory is that *the structural properties are implemented automatically by functoriality*. Because the type theory knows about the functorial action on transformations $\lesssim_{\mathsf{ctx}}$ of each type used in the definition, $\sigma \in -$ and $\times$ and Σ, the action of the inductive type

can be derived. Thus, we obtain weakening, exchange, and contraction for free. To handle substitution as well, we propose to investigate general conditions under which such a type determines a relative monad [7].

Generically equipping syntactic types like formula with the structural properties is possible using our previous work [38]. The real benefit of 2DTT is that it extends to representing the structural properties of proofs. After defining the syntax of propositions, the next step in programming with domain-specific logics is to specify a type $\Psi \vdash \phi$ which defines the truth of the proposition ϕ under assumptions Ψ. This type should also be structural in Ψ, but the definition of structurality is more subtle. For example, consider the rule of substitution:

$$\frac{\Psi \vdash e : \sigma \quad \Psi, x : \sigma, \Psi' \vdash \phi}{\Psi, \Psi'[e/x] \vdash \phi[e/x]}$$

This rule states that the substitution into the derivation shows that the substitution into the context entails the substitution into the proposition. To make this substitution principle come true, it must be the case that each inference rule of the logic commutes with substitution, in the sense that the substitution into the conclusion of the rule is the rule itself applied to the substitution into the premises.

2DTT elegantly accounts for exactly these two issues. First, proofs can be represented analogously to syntax, defining an indexed datatype $\mathsf{pf} : (\Sigma\, \psi{:}\mathsf{ctx}.\, \mathsf{formula}\ \psi) \to \mathsf{set}$, representing $\Psi \vdash \phi$ as terms of type $\mathsf{pf}\ (\Psi, \phi)$. Functoriality of Σ-types gives exactly the desired structural property, where e.g. substitution into the proof proves the substitution into the formula—illustrating the benefits of analyzing functoriality not just for simple type constructions but for dependent types like Σ and Π. Second, the fact that the subjects of judgements commute with substitution arises from the fact that all *terms*, as well as types, respect transformation: in addition to map, which states that substitution instances of types by transformable arguments are transformable, 2DTT contains rules stating that substitution instances of terms by transformable arguments are transformable.

Admissibility Once one has represented the syntax of propositions and proofs, dependent types allow the logic to be used to verify code—e.g. by annotating functions or computations (as in Ynot) with pre- and post-conditions. However, to know that this verification makes sense, it is necessary to prove properties of the logic, such as consistency; and, to make programming with logics practical, it is necessary to develop theorem provers that automatically discharge proof obligations. These tasks, mechanizing the metatheory of programming languages and logics, or, equivalently, computing with logical systems, require support for writing recursive functions that traverse syntax and proofs. For example:

$$\mathsf{consist} : \mathsf{pf}\ (\cdot, \bot) \to 0$$
$$\mathsf{prove} : \Pi\, \psi{:}\mathsf{ctx}.\, \Pi\, \phi{:}\mathsf{formula}\ \psi.\, \mathsf{pf}\ (\psi, \phi)\ \mathsf{option}$$

A consistency proof consist for a logic should have a type expressing that there are no closed proofs of falsehood. A theorem prover prove should have a type expressing that for any context and formula, the prover either returns a proof of the formula in that context, or fails.

However, rendering these types in 2DTT raises a subtle issue of what a function whose domain is a higher-dimensional type, such as $\Pi\, \psi{:}\mathsf{ctx}.\, A$, means. By analogy with type polymorphism, such a function may be *parametric*, if it behaves the same for all instantiations of α, or *ad-hoc*, if its behavior differs at different instances. Parametric quantification admits stronger reasoning principles, relating different instances by *naturality*. On the other hand, ad-hoc polymorphism admits more functions, by allowing more expressive elimination forms, including case-analysis and recursion on contexts and proofs. In 2DTT, functions of type $\Pi\, \psi{:}\mathsf{ctx}.\, A$ are necessarily parametric, as a consequence of a general principle that all terms respect transformation. However, there is a modality $!A$ that gives the underlying set of a structured type like ctx, and functions of type $\Pi\, \psi{:}!\mathsf{ctx}.\, A(\psi)$ are ad-hoc. We propose to investigate these two kinds of quantifiers, and their application to programming with logics.

Once we have investigated these issues, 2DTT will afford a language of signatures that significantly extends that of logical frameworks such as LF [30], because **representations of logics may exploit admissibility** (functional programs). This enables a wide class of more convenient representations of logics, encompassing negated premises in inference rules, infinitary sequent calculi for inductive types [31], and domain-specific languages and logics that inherit pattern-matching from the host language [71]. For example, a specification logic will often include inductive types or predicates. These can be represented conveniently and concisely if they are eliminated not by induction, but by an *ω-rule*. For example:

$$\frac{t : \mathsf{nat} \quad P(0) \quad P(1) \quad P(2) \quad \ldots}{P(t)}$$

To prove $P(t)$, it suffices to show $P(k)$ for each numeral k. The premise of this rule can be thought of as a function that delivers a proof of $P(k)$ for every k, and represented in 2DTT as follows:

$$\mathsf{omega} : (\Pi\, k{:}\mathsf{nat}.\, \mathsf{nd}\ (\psi, P[\mathsf{lit}(k)])) \to \mathsf{nd}\ (\psi, \forall P)$$

That is, to prove a predicate of all natural numbers, it suffices to show the predicate for each numeral in turn. This representation permits the logic to inherit induction and pattern-matching from the meta-language, which simplifies the development and leads to concise proofs of consistency via cut elimination [71].

References

[1] M. Abbott, T. Altenkirch, and N. Ghani. Containers: constructing strictly positive types. *Theoretic Computer Science*, 342(1):3–27, 2005.

[2] A. Abel. Miniagda: Integrating sized and dependent types. In A. Bove, E. Komendantskaya, and M. Niqui, editors, *Workshop on Partiality And Recursion in Interative Theorem Provers*, 2010.

[3] T. Altenkirch. Extensional equality in intensional type theory. In *IEEE Symposium on Logic in Computer Science*, 1999.

[4] T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl*, 2003.

[5] T. Altenkirch and P. Morris. Indexed containers. In *IEEE Symposium on Logic in Computer Science*, pages 277–285, Washington, DC, USA, 2009. IEEE Computer Society.

[6] T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In *CSL 1999: Computer Science Logic*. LNCS, Springer-Verlag, 1999.

[7] T. Altenkirch, J. Chapman, and T. Uustalu. Monads Need Not Be Endofunctors. In *Foundations of Software Science and Computational Structures*, volume 6014, chapter 21, pages 297–311. Springer Berlin Heidelberg, 2010.

[8] K. Avijit, A. Datta, and R. Harper. Distributed programming with distributed authorization. In *ACM SIGPLAN-SIGACT Symposium on Types in Language Design and Implementation*, 2010.

[9] S. Awodey and M. Warren. Homotopy theoretic models of identity types. *Mathematical Proceedings of the Cambridge Philosophical Society*, 2009.

[10] E. S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70:35–64, January 1990.

[11] G. Bellè, C. Jay, and E. Moggi. Functorial ML. In H. Kuchen and S. Doaitse Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 1140 of *Lecture Notes in Computer Science*, pages 32–46. Springer Berlin / Heidelberg, 1996.

[12] F. Bellegarde and J. Hook. Substitution: A formal methods case study using monads and transformations. *Science of Computer Programming*, 23(2–3):287–311, 1994.

[13] M. Benke, P. Dybjer, and P. Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10(4):265–289, 2003.

[14] J.-P. Bernardy, P. Jansson, and R. Paterson. Parametricity and dependent types. In *ACM SIGPLAN International Conference on Functional Programming*, 2010.

[15] R. S. Bird and R. Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, 1999.

[16] L. Cardelli. Notes about $F^{\omega}_{<:}$. Unpublished., 1990.

[17] A. Chlipala, G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. In *icfp*, 2009.

[18] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice Hall, 1986.

[19] Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.2*. INRIA, 2009. Available from `http://coq.inria.fr/`.

[20] R. D. Cosmo. *Isomorphisms of types: from lambda-calculus to information retrieval and language design*. Birkhauser, 1995.

[21] N. G. de Bruijn. A lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[22] H. DeYoung, D. Garg, and F. Pfenning. An authorization logic with explicit time. In *IEEE Computer Security Foundations Symposium*, 2008.

[23] D. Duggan and A. Compagnoni. Subtyping for object type constructors. In *Workshop On Foundations Of Object-Oriented Languages*, 1999.

[24] M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *IEEE Symposium on Logic in Computer Science*, 1999.

[25] N. Gambino and R. Garner. The identity type weak factorisation system. *Theoretical Computer Science*, 409(3):94–109, 2008.

[26] D. Garg. *Proof Theory for Authorization Logic and Its Application to a Practical File System.* PhD thesis, Carnegie Mellon University, 2009.

[27] D. Garg and F. Pfenning. A proof-carrying file system. In *IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[28] R. Garner. Two-dimensional models of type theory. *Mathematical. Structures in Computer Science*, 19(4):687–736, 2009.

[29] M. Grandis. *Directed Algebraic Topology: Models of non-reversible worlds*. Cambridge University Press, 2009.

[30] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1), 1993.

[31] D. Hilbert. Die grundlegung der elementaren zahlenlehre. *Mathematische Annalen*, 104:485–494, 1931.

[32] M. Hofmann. *Extensional Concepts in Intensional Type Theory*. PhD thesis, University of Edinburgh, 1995.

[33] M. Hofmann. Semantical analysis of higher-order abstract syntax. In *IEEE Symposium on Logic in Computer Science*, 1999.

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

[35] L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, , and S. Zdancewic. Aura: A programming language for authorization and audit. In *ACM SIGPLAN International Conference on Functional Programming*, 2008.

[36] R. Laemmel and S. Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *ACM SIGPLAN-SIGACT Symposium on Types in Language Design and Implementation*, 2003.

[37] D. R. Licata. *Dependently Typed Programming with Domain-Specific Logics*. PhD thesis, Carnegie Mellon University, 2011. Available from `http://www.cs.cmu.edu/~drl/pubs/thesis/thesis.pdf`.

[38] D. R. Licata and R. Harper. A universe of binding and computation. In *ACM SIGPLAN International Conference on Functional Programming*, 2009.

[39] D. R. Licata and R. Harper. A monadic formalization of ML5. In *Workshop on Logical Frameworks and Meta-languages: Theory and Practice*, July 2010.

[40] D. R. Licata, N. Zeilberger, and R. Harper. Focusing on binding and computation. In *IEEE Symposium on Logic in Computer Science*, 2008.

[41] P. L. Lumsdaine. Weak ω-categories from intensional type theory. In *International Conference on Typed Lambda Calculi and Applications*, 2009.

[42] G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a verified relational database management system. In *popl*, 2010.

[43] P. Martin-Löf. An intuitionistic theory of types: Predicative part. In H. Rose and J. Shepherdson, editors, *Logic Colloquium*. Elsevier, 1975.

[44] P. Martin-Lf. Analytic and synthetic judgements in type theory. In *Kant and Contemporary Epistemology*, pages 87–99. Kluwer, 1996.

[45] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 15(1), 2004.

[46] J. Morgenstern and D. R. Licata. Security-typed programming within dependently-typed programming. In *ACM SIGPLAN International Conference on Functional Programming*, 2010.

[47] T. Murphy VII. The wizard of TILT: Efficient, convenient and abstract type representations. Technical Report CMU-CS-02-120, School of Computer Science, Carnegie Mellon University, 2002. URL `http://www.cs.cmu.edu/~tom7/papers/`.

[48] T. Murphy VII. *Modal Types for Mobile Code*. PhD thesis, Carnegie Mellon, January 2008. Available as technical report CMU-CS-08-126.

[49] T. Murphy VII, K. Crary, R. Harper, and F. Pfenning. A symmetric modal lambda calculus for distributed computing. In H. Ganzinger, editor, *IEEE Symposium on Logic in Computer Science*, pages 286–295. IEEE Press, 2004.

[50] T. Murphy VII, K. Crary, and R. Harper. Distributed control flow with classical modal logic. In *Computer Science Logic*, volume 3634 of *Lecture Notes in Computer Science*, pages 51–69. Springer-Verlag, 2005.

[51] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the awkward squad. In *ACM SIGPLAN International Conference on Functional Programming*, 2008.

[52] R. Nederpelt, J. Geuvers, and R. de Vrijer, editors. *Selected Papers on AUTOMATH*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1994.

[53] B. Nordström, K. Peterson, and J. Smith. *Programming in Martin-Löf's Type Theory, an Introduction*. Clarendon Press, 1990.

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

[55] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In H. Ganzinger, editor, *International Conference on Automated Deduction*, pages 202–206, 1999.

[56] B. Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 371–382, 2008.

[57] G. D. Plotkin and M. Abadi. A logic for parametric polymorphism. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 361–375, London, UK, 1993. Springer-Verlag.

[58] A. Poswolsky and C. Schürmann. Practical programming with higher-order encodings and dependent types. In *European Symposium on Programming*, 2008.

[59] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science*, 2002.

[60] R. Seely. Modeling computations: a 2-categorical framework. In *IEEE Symposium on Logic in Computer Science*, pages 65–71, 1987.

[61] M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. In *Symposium on Foundations of Computer Science*, pages 13–17, Washington, DC, USA, 1977. IEEE Computer Society.

[62] M. Steffen. *Polarized Higher-Order Subtyping*. PhD thesis, Universitaet Erlangen-Nuernberg, 1998.

[63] N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in Fine. In *European Symposium on Programming*, 2010.

[64] B. van den Berg and R. Garner. Types are weak ω-groupoids. Available from `http://www.dpmms.cam.ac.uk/~rhgg2/Typesom/Typesom.html`, 2010.

[65] V. Voevodsky. The equivalence axiom and univalent models of type theory. Available from `http://www.math.ias.edu/~vladimir/Site3/home_files/`, 2010.

[66] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1987.

[67] P. Wadler. Theorems for free! In *International Conference on Functional Programming and Computer Architecture*, 1989.

[68] M. Wand. Fixed-point constructions in order-enriched categories. *Theoretical Computer Science*, 8 (1):13 – 30, 1979.

[69] M. A. Warren. *Homotopy theoretic aspects of constructive type theory*. PhD thesis, Carnegie Mellon University, 2008.

[70] R. Wisnesky, G. Malecha, and G. Morrisett. Certified web services in ynot. In *International Workshop on Automated Specification and Verification of Web Systems*, 2009.

[71] N. Zeilberger. Focusing and higher-order abstract syntax. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 359–369, 2008.