Abstract: Dependently Typed Programming with Domain-Specific Logics

Daniel R. Licata

Introduction

This dissertation describes progress on programming with domain-specific specification logics in dependently typed programming languages. Domain-specific logics are a promising way to verify software, using a logic tailored to a style of programming or an application domain. Examples of domain-specific logics include separation logic, which has been used to verify imperative programs, and authorization logics, which have been used to verify security properties in security-typed languages. The first goal of the research described here is to show that it is possible to define, study, automate, and use domain-specific logics within a dependently typed programming language. We demonstrate this fact with a significant new example, showing how to embed a security-typed language using dependent types.

This example suggests that better support for programming with logics in type theory will facilitate this style of program verification. The central notion in logic is consequence—entailment from premises to conclusions—and two notions of consequence are necessary for programming with logics: derivability, which captures uniform reasoning, and admissibility, which captures inductive proofs and functional programs. Presently, derivability is better supported in LF-based proof assistants, such as Twelf, Delphin, and Beluga, whereas admissibility is better supported in proof assistants based on Martin-Löf type theory, such as Coq, Agda, and Epigram. Our second contribution is to show that it is possible to implement, within a dependently typed programming language, a logical framework that allows derivability and admissibility to be mixed in novel and interesting ways.

The above framework is simply-typed, which makes it suitable for programming with abstract syntax but not logical derivations. Our third contribution is to generalize this framework to dependent types, which we accomplish as an instance of a more general problem: We describe Directed Type Theory (DTT), a new notion of dependent type theory, inspired by higher-dimensional category theory, which equips each type with a notion of transformation on its elements. The structural properties of a logic arise as a special case, by considering a type of contexts equipped with an appropriate notion of transformation.

DTT is an exciting development independently of our application, as it contributes to recent connections between type theory, homotopy theory, and category theory, which have led to the development of a subject called Homotopy Type Theory. Since the publication of this dissertation, the formulation of Directed Type Theory described here led to the first computational interpretation for a fragment of Homotopy Type Theory (Licata and Harper, 2012). Moreover, this computational perspective on Homotopy Type Theory led to the development of type-theoretic techniques for proving theorems in the mathematical discipline of homotopy theory. Using these techniques, the author and others have given computer-checked proofs of a number of theorems in homotopy theory. These include calculations of homotopy groups of spheres, such as $\pi_1(S^1)$ (Licata and Shulman, 2013), $\pi_n(S^n)$, and the Freudenthal Suspension theorem; a construction of Eilenberg-Mac Lane spaces; and van Kampen's theorem.

Part I: Programming with Logics in Existing Languages

Different programs and correctness properties require different logics tailored to reasoning about them: For example, Hoare logic and separation logic (Reynolds, 2002) are used to reason about imperative programs which manipulate the state of memory in intricate ways. Temporal logics (Clarke et al., 2002) are used to reason about the evolution of concurrent processes. Authorization logics (Abadi et al., 1993) are used to verify security properties. Differential dynamic logic is used to reason about hybrid (discrete/continuous) cyber-physical systems (Platzer, 2010). Because different logics are appropriate for different tasks, specification logic design necessarily becomes part of the programming process.

What are the typical activities involved in designing a new logic? The first task is to define the logic, and write down the syntax of its propositions and proof rules. This must include a way of integrating the logic with the programming language, either by allowing logical formulas that refer to programs, as in a specification logic, or by externally assigning propositions to programs, as in a type system. To show that the logic is reasonable, it is common to prove some sort of correctness result about the logic: one may prove that the logic is consistent, or more generally that it ensures that programs have the properties of interest. Such proofs may use syntactic methods (structural induction on derivations), or semantic ones (showing that the logic is sound and/or complete with respect to a notion of truth). Next, to make verification practical, it is common to implement some sort of automated or interactive theorem proving for the logic.

At a high level, the first goal of the research described here is to advance the idea that

> *It is possible to define, study, automate, and use domain-specific logics within a dependently typed programming language.*

Dependent types make logic design part of the programming process, so that programmers can define logics and use them both to reason about their code and to explain it to others. The result is *code that tells you why it works*. All of the tasks involved in programming with logics can be carried out within a dependently typed host language, which improves on current practice, where it is common to implement logics and type systems either within language implementations or as external tools. Dependently typed languages provide richer tools for describing, studying, and implementing logics than the current simply-typed languages used in these implementations, and they permit program verification as well.

Though the particular approach that we advocate has some modern twists, this general approach to program verification is an old idea and it is only through decades of work on dependently typed proof assistants and programming languages that it is now becoming a reality. In the first of this dissertation, we contribute some new examples as further evidence of the above claim. Our most significant new example shows how to do security-typed programming, in the style of the languages PCML5 (Avijit et al., 2010), Aura (Jia et al., 2008), and Fine (Swamy et al., 2010), within a dependently typed programming language. Second, we show how to represent and implement the semantics of Reed and Pierce (2010)'s type system for differential privacy (Dinur and Nissim, 2003; Dwork and Nissim, 2004; Dwork et al., 2006), which provides an extensible approach to implementing differentially private algorithms in a formally certified manner. Throughout the dissertation, we use the dependently typed programming language Agda (Norell, 2007).

We describe the first example, of security-typed programming, in more detail here. Security-typed programming languages allow programmers to specify and enforce security policies, which describe both *access control*—who is permitted to access sensitive resources?—and *information flow*—what are they permitted to do with these resources once they get them? In this part of the dissertation, we implement a library, Aglet, which accounts for the major features of existing security-typed programming languages, such as Aura (Jia et al., 2008), PCML5 (Avijit et al., 2010), and Fine (Swamy et al., 2010). These include:

Decentralized access control, where access control policies are expressed as propositions in an *authorization logic*, Garg and Pfenning's BL_0 (Garg, 2009). This permits *decentralized* access control policies, expressed as the aggregate of statements made by different principals about the resources they control. In our embedding, we represent BL_0's propositions and proofs using dependent types, and exploit Agda's type checker to validate the correctness of proofs.

Dependently typed PCA,where primitives that access resources, such as file system operations, require programmers to provide a proof of authorization, which is guaranteed by the type system to be a well-formed proof of the correct proposition. *Ephemeral and dynamic policies:* Whether or not one may access a resource is often dependent upon the state of a system. For example, in a conference management server, authors may submit a paper, but only before the submission deadline. Following Hoare Type Theory (Nanevski et al., 2008), we define a type $\bigcirc\ \Gamma\ A\ \Gamma'$, which represents a computation that, given precondition Γ, returns a value of type A, with postcondition Γ'. Here, Γ and Γ' are propositions from the authorization logic, describing the state of resources in the system.

Compile-time and Run-time Theorem Proving: Dependently typed PCA admits a sliding scale between static and dynamic verification. At the static end, one can verify, at compile-time, that a program complies with a statically-given authorization policy. This verification consists of annotating each access to a resource with an authorization proof, whose correctness is ensured by type checking. However, in many programs, the policy is not known at compile time—e.g., the policy may depend upon a system's state. Such programs may dynamically test whether each operation is permitted before performing it, in which case dependently typed PCA ensures that the correct dynamic checks are made and that failure cases are handled. A program may also mix static and dynamic verification: for example, a program may dynamically check that an expected policy is in effect, and then, in the scope of that check, deduce consequences statically. Security-typed languages use theorem provers to reduce the burden of static proofs (as in Fine) and to implement dynamic checks (as in PCML5). We have implemented a certified theorem prover for BL_0, based on a focused sequent calculus. Our theorem prover can be run at compile-time and at run-time, fulfilling both of these roles. The theorem prover also saves programmers from having to understand the details of the authorization logic, as they often do not need to write proofs manually.

Part II: Mixing Derivability and Admissibility

Of course, we would like to achieve a state of affairs where it is not just possible, but *practical*, to program with logics using dependent types. At present researchers can, with some effort, do this type of programming, but we have not achieved the practicality necessary for wide-spread adoption. The remaining parts of this thesis make some technical contributions towards making programming with logics easier.

A key notion in logic is the *hypothetical judgement*, which codifies reasoning from assumptions, satisfying certain structural properties (identity, substitution, weakening, exchange, contraction). Any tool for programming with logics must support two notions of hypothetical judgement, *derivability* ($\vdash$), which captures uniform reasoning, and *admissibility* ($\vDash$), which captures inductive proofs and functional programs. These notions are essential for describing and programming with logics. Derivability is better supported in proof assistants that use type theory in the style of LF (Harper et al., 1993), whereas admissibility is better supported in proof assistants that use type theory in the style of Martin-Löf type theory (MLTT) (Martin-Löf, 1975).

The examples in Part I motivate providing a generic implementation of derivability inside of MLTT. One option would be to implement a well-known logical framework, such as LF. However, this would not account for inductive definitions that mix derivability and admissibility assumptions. Admissibility premises are useful because they permit infinitely branching derivations, and allow certain forms of *side conditions*, such as negated premises (as $J \vDash \mathsf{false}$). Thus, we instead take the opportunity to explore a new logical framework that allows such interaction. In this chapter, we focus on the hypothetical judgement, precluding dependency on assumptions. Because of this restriction to "non-dependent judgements," our examples focus on simply-typed programming with abstract syntax. Our goal is to demonstrate that

> *It is possible to implement, within a dependently typed programming language, a logical framework that allows derivability and admissibility to be mixed in novel and interesting ways.*

The reason that mixing admissibility and derivability is difficult is that rules with admissibility premises are not necessarily pure, and, consequently, the structural properties do not necessarily hold. For example, in logical frameworks such as LF, it is always possible to weaken a value of type J to $L \vdash J$. However, this is not necessarily possible when J itself is an admissibility. The reason is that we interpret an admissibility in a derivability context, $\Psi \vdash (J \vDash K)$, as essentially the same thing as an admissibility between derivabilities: $(\Psi \vdash J) \vDash (\Psi \vdash K)$. Now, suppose we are given a function f of type $\Psi \vdash (J \vDash K)$, and we try to weaken this to a function of type $(\Psi, L) \vdash (J \vDash K)$. This requires an admissibility function from $(\Psi, L) \vdash J$ to $(\Psi, L) \vdash K$. Since f is a black box, we can only hope to achieve this by pre- and post-composing with appropriate functions. The post-composition must take $\Psi \vdash K$ to $\Psi, L \vdash K$, which is a recursive application of weakening. However, the pre-composition has a contravariant flip: we require *strengthening* $(\Psi, L) \vdash J$ to $\Psi \vdash J$ in order to call f. Such a strengthening function does not in general exist, because the derivation of J might be that assumption of L. Similarly, substitution of terms for variables is not necessarily possible, because substitution requires weakening.

As a concrete example, consider a closed admissibility function of type $\cdot \vdash (\mathsf{exp} \vDash \mathsf{exp})$, which is defined by case-analysis over closed λ-calculus expressions, giving cases for functions and applications—but *not* for variables, because there are no variables in the empty context. Weakening such a function to type $\mathsf{exp} \vdash (\mathsf{exp} \vDash \mathsf{exp})$ *enlarges* its domain, asking it to handle cases that it does not cover. Another example, which is particularly stark, is weakening $L \vDash \mathsf{false}$, which refutes the existence of any terms of type L, to $L \vdash (L \vDash \mathsf{false})$, which has one such term to account for.

Our solution to this problem rests on the observation that there is nothing about the inductive definition of derivability that requires the structural properties to hold: the identity principle is taken as a rule, but the remaining structural properties are proved admissible. Thus, we may give a definition of a framework that allows both admissibility and *pre-derivability*. Pre-derivability captures the notion of a scoped assumption, which can be used via the identity principal, but does not necessarily satisfy the remaining structural properties. Then, we analyze circumstances under which the structural properties hold, in which case pre-derivability is actually derivability. In programming terms, this amounts to implementing the structural properties as *generic programs*, based on some conditions on the type involved.

Higher-order Focusing We were originally motivated to consider these interactions by studying derivability and admissibility in the context of Zeilberger's higher-order focusing (Zeilberger, 2008a,b, 2009). Zeilberger's previous work showed how to do higher-order focusing for classical logic, and for the "positively-only" part of intuitionistic logic. Our work in this part also includes a formulation of higher-order focusing for full intuitionistic propositional logic, which is of interest independently of the application.

The key idea in higher-order focusing is to use admissibility functions in the meta-language to represent inversion phases. First, a type is specified by *patterns*, which give the shape of a focus phase. Second, an inversion phase is represented by a function from patterns to proofs, which delivers a proof of the conclusion for each pattern of the assumption being inverted. The advantage of higher-order focusing for our purposes is that the logic is parametrized by the notion of admissibility function used to witness inversions. This open-endedness gives us space to implement the structural properties of pre-derivability by recursion over patterns in the meta-language, and to internalize these as inversions.

A Universe in Agda We later realized that the same idea of generic programming can be applied within existing proof assistants. This led to the Agda implementation of a logical framework with two function-like type constructors, $\supset$ (for admissibility) and $\Rightarrow$ (for pre-derivability). $A \supset B$ classifies Agda functions, while $D \Rightarrow A$ classifies *values of type A with a free scoped assumption of type D*. In some cases, $D \Rightarrow A$ represents a derivability, and therefore determines a function given by substitution, but in some cases it does not. However, rather than leaving it to the programmer to implement the structural properties, we observe that they *are* in fact definable generically, not for every type $D \Rightarrow A$, but under certain conditions on the types D and A. For example, returning to our failed attempt to weaken $A \supset B$ above, if variables of type D could never appear in terms of type A, then the required strengthening operation would exist. As a rough rule of thumb, one can weaken with types that do not appear to the left of a computational arrow in the type being weakened, and similarly for substitution. Our framework implements the structural properties generically but conditionally, providing programmers with the structural properties "for free" in many cases. We implement $\Rightarrow$ with well-scoped de Bruijn indices, but another first-order representation of derivability with explicit contexts could be used instead.

Our framework is implemented as a universe in Agda. This means that we (a) give a syntax for the types of the framework and (b) give a function mapping the types of our language to certain Agda types; the programs of the framework are then the Agda programs of those types. This implementation strategy allows us to reuse the considerable implementation effort that has gone into Agda, and to exploit generic programming within dependently typed programming (Altenkirch and McBride, 2003) to implement the structural properties. Additionally, it permits programs written using our framework to interact with existing Agda code.

In particular, we define a universe of *contextual* types, which permit inference rules to be written in a *local* form: the context of scoped assumptions need not be mentioned explicitly in rules, but is passed in from the outside. This often permits more concise specifications. Semantically, a type in the universe is interpreted as a function from contexts to Agda types. For example, the contextual type $A \otimes B$, is interpreted as the function $\lambda\Psi.\ \langle\Psi\rangle\ A \times \langle\Psi\rangle\ B$, where $\langle\Psi\rangle\ A$ stands for the interpretation of A at Ψ, and $\times$ is pairing in Agda. The universe types are thus "point-free" combinators that generate functions from contexts to types, without mentioning the context argument explicitly. For example, $D \Rightarrow A$ is interpreted as $\lambda\Psi.\ \langle\Psi, D\rangle\ A$—it extends the current context without mentioning it by name. That said, our framework supplies a rich enough set of combinators that contexts can be mentioned explicitly when this is helpful—for example, the contextual type $[\Psi]A$ represents a constant function $\lambda\_.\ \langle\Psi\rangle\ A$ which interprets A relative to the given Ψ, ignoring the context supplied by the interpretation.

Our framework implements a variety of structural properties for the universe, including weakening, substitution, exchange, contraction, and subordination-based strengthening (Virga, 1999), all using a single generic `map` function for datatypes that mix binding and computation. The structural properties' preconditions are defined computationally, so that our framework can discharge these conditions automatically in many cases. This gives the programmer free access to weakening, substitution, etc. (when they hold). For example, we implement normalization-by-evaluation (Berger and Schwichtenberg, 1991; Martin-Löf, 1975) for the untyped λ-calculus, an example considered in FreshML by Shinwell et al. (2003). Our version of this algorithm makes essential use of a datatype mixing binding and computation, and our type system verifies that evaluation maps closed terms to closed terms.

Part III: Directed Dependent Type Theory

The dependently typed analogue of the hypothetical judgement is called the generic judgement, and implementing a logical framework with genericity is not as easy. The difficulty comes in implementing the structural properties of the generic judgement. We analyze these difficulties as an instance of a much more general phenomenon, that of dependently typed programming with *directed types* in *higher-dimensional type theory*.

Higher-dimensional type theory

One of the most cherished ideas in programming languages is the correspondence between logic, type theory, and category theory. Recent research has suggested a second holy trinity, between dependent type theory, higher-dimensional category theory, and homotopy theory (Awodey and Warren, 2009; Garner, 2009; Hofmann and Streicher, 1998; Lumsdaine, 2009; van den Berg and Garner, 2011; Voevodsky, 2010). The homotopy hypothesis (Baez and Shulman, 2007) postulates a correspondence between higher-dimensional categories and higher-dimensional homotopy types, and recent work has extended this to a correspondence with type theory.

On the type theoretic side, this correspondence suggests enriching dependent type theories with *higher-dimensional types*. Type theory is traditionally thought of as talking about sets, which are types of dimension 0: sets have elements (objects, or 0-cells), which may or may not be equal. Types of dimension 1 are like categories: they have elements, but also have a notion of map between elements (morphisms, or 1-cells), and every function on such a type must respect this notion of map. A setoid—a set equipped with an equivalence relation—is an example of a 1-dimensional type, but in general the notion of morphism can have computational content. An example is a category of sets with isomorphisms as morphisms—in this case, there can be many distinguishable maps between two sets. The type of contexts and structural properties is 1-dimensional as well. Types of dimension 2 have elements, maps, and maps between maps (2-cells)—an example is the collection of all categories, with functors as maps and natural transformations as maps between maps. This is an example of a *2-category*. In general, an n-dimensional type corresponds to an *n-category*, or, equivalently, a homotopy n-type.

One important reason to identify the dimension of a type is that different notions of equality are appropriate for types of different dimensions. For sets (0-types), there is nothing but equality of elements. For categories (1-types), elements can be equal, but they might also be *isomorphic*, if there are maps (1-cells) between them that compose to the identity. For categories, the appropriate notion of equality is *equivalence*: maps back and forth, whose compositions are isomorphic to the identity, using the 2-cells (two categories are equivalent if there are functors back and forth that are naturally isomorphic to the identity). A goal of higher-dimensional type theory is to provide support for n-equivalence in full generality: n-equivalent terms should be provably equal in the theory, and all families of types and objects should respect n-equivalence. Voevodsky's univalence axiom (Voevodsky, 2010) postulates that *equivalent types are interchangable*, where equivalence is interpreted as appropriate for the dimension. On the category-theoretic and homotopy-theoretic side, this correspondence suggests using a higher-order type theory as a meta-language, both for category theory and homotopy theory themselves, and through them for all formalized mathematics—as has recently been espoused by Voevodsky (Voevodsky, 2010). In such a theory, mathematicians will always be able to reason up to equivalence, no matter the appropriate notion of equivalence for the objects in question.

What is perhaps surprising is that Martin-Löf's *intensional* type theory is in fact sound for higher-dimensional types, with equivalence represented by the identity type $\mathsf{Id}_A\, M N$: there is nothing in intensional type theory that restricts the dimension of a type. The identity type is eliminated by subst

$$\frac{x{:}A \vdash C\ \mathsf{type} \quad P : \mathsf{Id}_A\ M\ N \quad Q : C[M/x]}{\mathsf{subst}_C\ P\ Q : C[N/x]}$$

(or, more generally, the J rule, which allows a motive C that is dependent on the equivalence P as well). This rule says that any dependent type respects equivalence, which is true in any dimension.

Higher-dimensional interpretations were pioneered in Hofmann and Streicher's groupoid interpretation (Hofmann and Streicher, 1998), which interprets intensional type theory in the 2-category of groupoids, functors, and natural transformations: A closed type denotes a groupoid, whose objects are the terms of the type, and whose morphisms are the (1-)equivalences between them. Identity and composition ensure that propositional equality is reflexive and transitive, while the groupoid structure, which demands that every map be invertible, ensures symmetry. The hom-sets of a type A are presented in the syntax as the inhabitants of the identity type $\mathsf{Id}_A\ M\ N$. Open types and terms are interpreted as functors, whose object parts are (roughly) the usual sets and elements of the set-theoretic semantics, and whose morphism parts show how those types and terms preserve equivalence. When the context is not a discrete groupoid, the functorial action on equivalences is computationally relevant. For example, consider a universe set of sets and isomorphisms, by taking the propositional equalities between S_1 and S_2 to be invertible functions $\mathsf{El}(S_1) \rightarrow \mathsf{El}(S_2)$. The rule subst states that all type families respect isomorphism: for any $x : \mathsf{set} \vdash C : \mathsf{set}$, $A \cong B$ implies $C[A] \cong C[B]$. Computationally, the lifting of the isomorphism is given by the functorial action of the type family C.

Recent work has generalized this to higher dimensions. On the categorical side, Garner (2009) generalizes the groupoid interpretation to a class of 2-categories, rather than just *Groupoid*. Lumsdaine (2009) and van den Berg and Garner (2011) show that the syntax of intensional type theory forms a weak ω-category. On the homotopy-theoretic side, Awodey and Warren (2009) show how to interpret intensional type theory into abstract homotopy theory (Quillen model categories).

Structural Properties as Functoriality

Altenkirch and Reus (1999); Fiore et al. (1999); Hofmann (1999) show that the structural properties of hypothetical judgements can be viewed as functoriality. Let's put this observation to use in our setting: The logical framework from Part II, the universe of contextual Types, are interpreted as functions $\mathsf{Ctx} \rightarrow \mathsf{Set}$. Our first observation is that the structural properties endow these functions with the structure of a *functor* from a category of contexts to the category of sets. To cover the structural properties of weakening, exchange, and contraction, we can define Ctx to be the category where an object is a context Ψ, and a morphism $\Psi_1 \longrightarrow \Psi_2$ is a variable-for-variable substitution, represented using the type $\Psi_1 \subseteq \Psi_2$ defined above. A functor $Ctx \longrightarrow Sets$ consists of a family of sets F_Ψ, which preserves context maps: if $\Psi \subseteq \Psi'$ then there is a function $F_\Psi \rightarrow F_{\Psi'}$—which is exactly the definition of weakening (for a whole context at once).

The Types of our universe show that the collection of functors $Ctx \longrightarrow Sets$ is closed under various type-forming operations, whose action on morphisms is given in the definition of map. For example, $F \otimes G$ is the product functor, whose action on an object Ψ is given by the product of F_Ψ and G_Ψ, and action on morphisms is given componentwise. The complication is that the type $F \supset G$ defines an action on objects (Ψ goes to $F_\Psi \rightarrow G_\Psi$), but is *not* functorial, because the $\rightarrow$ is *contravariant* in its domain. Given a contravariant functor $F : Ctx^{\mathsf{op}} \longrightarrow Sets$ and a covariant functor $G : Ctx \longrightarrow Sets$, we can form $F \supset G$ with action on morphisms given by pre- and post-composition. However, given a covariant functor for the domain, then the intended pre-composition faces the wrong direction (we have $F_{\Psi_1} \rightarrow F_{\Psi_2}$ when we need $F_{\Psi_2} \rightarrow F_{\Psi_1}$). This provides another explanation for why the structural properties fail for admissibility functions: the structural properties are implemented by the functorial actions of the type constructors, but $\supset$ does not determine a functor.

Thus, we can view the generic programming implementation of the structural properties as logical framework as establishing conditions under which the types denote functors.

To make the embedded logical framework described above useful for programming with logics, rather than just syntax, we need general, as well as hypothetical, judgements. General judgements model variables, as in a first-order quantifier. However, the structural properties of the generic judgements are more subtle than those of the hypothetical; for example, the rule for substitution:

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau, \Gamma' \vdash J}{\Gamma, \Gamma'[e/x] \vdash J[e/x]}\ \mathsf{subst}$$

This rule says that the substitution into the derivation proves the substitution into the context and judgement. When implemented with de Bruijn indices, weakening, exchange, and contraction exhibit similar phenomena: stating them for derivations requires a corresponding action on judgements.

The categorical account of syntax with binding can be extended to this case. Indeed, an indexed judgement like $\Gamma \vdash A\,\mathsf{true}$ determines a functor $(\Sigma\ \Psi{:}Ctx.\ \mathsf{Prop}(\Psi)) \longrightarrow Sets$, where Σ is modeled categorically by the Grothendieck construction and gives exactly to the structural properties of the generic judgement: the substitution into the derivation proves the substitution into the context and judgement.

Directed Type Theory

In Part III of this dissertation, we use the tools provided by higher-dimensional type theory to design a logical framework that allows mixing admissibility and derivability for generic judgements. The structural properties of the generic judgement are instance of higher-dimensional structure, given by entailment between logical propositions.

However, there is a major complication: entailment is not symmetric, and thus is not an instance of the higher-dimensional theories discussed above, which account only for symmetric notions of equivalence. This application prompted us to begin to study a new notion of *directed dependent type theory* which accounts for asymmetric or directed types. This requires not just a new semantic interpretation, but also a new proof theory. In this part of the dissertation, we study the 2-dimensional case, which accounts for types of level 2, and give it a semantics in category theory. We show that

> *A language with directed types provides a useful framework for describing the structural properties of the generic judgement.*

Directed type theory is of interest independently of our application. First, it extends the correspondence between type theory, ω-groupoids, and homotopy theory to directed type theory, ω-categories, and directed homotopy theory. A directed type theory would provide a more general meta-language for higher category theory and homotopy theory, allowing the expression of non-symmetric notions. While directed homotopy theory is not as well studied, it may have applications in physics, due to the directed aspects of space-time. On a more practical level, programming language semantics is full of directed notions—reduction, monotone functions on domains—and directed type theory may have applications in mechanization of these languages. It may also shed light on the problem of combining dependent types with concurrency, which has been analyzed in homotopy-theoretic terms (Gaucher, 2003).

Another reason to study directed type theory is that the familiar dependent type constructors have an action on directed transformations, and with a few simple changes to the theory we can expose this nature. Relaxing symmetry requires two main changes to the type theory: First, the type theory makes the *variances* of type families explicit, so that we have the language to say, e.g., that Π is contravariant in its domain, but covariant in its range. Second, the type theory exposes higher-dimensional structure at the judgemental level, not through a propositional equality type. The reason for this is that a propositional equality type, in its standard formulation, relies subtly on the symmetry of equivalence when proving that the type $\Gamma \vdash \mathsf{Id}_A\ M\ N\ \mathsf{type}$ is functorial in Γ. One solution to this problem, which we discuss some, is to investigate *directed hom types* that relate two terms of opposite variance. However, a more fundamental approach is to express the structure we want—transformations between two terms of the same variance—as a judgemental notion, which is not required to be functorial.

In this part, we take a first step in the investigation of directed type theory, considering the simplest non-trivial case, that of a 2-dimensional theory. *2-dimensional directed dependent type theory*, or *2DTT*, has three layers: types (0-cells), terms (1-cells), and transformations (2-cells), and can be interpreted in a 2-category. Each type itself denotes a 1-category, with terms as objects and transformations as morphisms. For simplicity, the syntax reflects the structure of a strict 2-category, where the associativity and unit laws of 1-cells are definitional equalities, not a weak 2-category (bicategory), where they hold only up to equivalence. Additionally, we treat *equality* (but not equivalence) as in extensional type theory. We validate 2DTT by an interpretation in the strict 2-category Cat of categories, functors, and natural transformations, generalizing the groupoid interpretation—though we conjecture that this semantics should extend to a class of 2-categories with the appropriate structure.

In the same way that symmetric higher-dimensional type theory makes a dependently typed programming language out of groupoids, 2DTT makes a programming language out of general categories. Because this includes dependent type constructors like Π and Σ, equipped with their functorial actions on morphisms, a programmer can derive the structural properties of the generic judgement from a datatype definition, by functoriality—even when the datatype includes admissibility functions represented using Π.

References

M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

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

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

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.

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

J. C. Baez and M. Shulman. Lectures on n-categories and cohomology. Available from `http://arxiv.org/abs/math/0608420v2`, 2007.

U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *IEEE Symposium on Logic in Computer Science*, 1991.

E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2002.

I. Dinur and K. Nissim. Revealing information while preserving privacy. In *ACM Symposium on Principles of Database Systems*, 2003.

C. Dwork and K. Nissim. Privacy-preserving datamining on vertically partitioned databases. In *International Cryptology Conference*, 2004.

C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Theoretical Cryptography Conference*, 2006.

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

D. Garg. Proof search in an authorization logic. Technical Report CMU-CS-09-121, Computer Science Department, Carnegie Mellon University, April 2009.

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

P. Gaucher. A model category for the homotopy theory of concurrency. *Homology, Homotopy, and Applications*, 5(1): 549–599, 2003.

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

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

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

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.

D. R. Licata and R. Harper. Canonicity for 2-dimensional type theory. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.

D. R. Licata and M. Shulman. Calculating the fundamental group of the cirlce in homotopy type theory. In *IEEE Symposium on Logic in Computer Science*, 2013.

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

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

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.

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

A. Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, Heidelberg, 2010.

J. Reed and B. C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *ACM SIGPLAN International Conference on Functional Programming*, 2010.

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

M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *ACM SIGPLAN International Conference on Functional Programming*, pages 263–274, August 2003.

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

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

R. Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Carnegie Mellon University, 1999.

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

N. Zeilberger. On the unity of duality. *Annals of Pure and Applied Logic*, 153(1–3), 2008a. Special issue on "Classical Logic and Computation".

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

N. Zeilberger. *The logical basis of evaluation order and pattern matching*. PhD thesis, Carnegie Mellon University, 2009.