Abstraction and Invariance for Algebraically Indexed Types

Robert Atkey Patricia Johann
University of Strathclyde
{Robert.Atkey,Patricia.Johann}@strath.ac.uk

Andrew Kennedy
Microsoft Research Cambridge
akenn@microsoft.com

Abstract

Reynolds' relational parametricity provides a powerful way to reason about programs in terms of invariance under changes of data representation. A dazzling array of applications of Reynolds' theory exists, exploiting invariance to yield "free theorems", non-inhabitation results, and encodings of algebraic datatypes. Outside computer science, invariance is a common theme running through many areas of mathematics and physics. For example, the area of a triangle is unaltered by rotation or flipping. If we scale a triangle, then we scale its area, maintaining an invariant relationship between the two. The transformations under which properties are invariant are often organised into groups, with the algebraic structure reflecting the composability and invertibility of transformations.

In this paper, we investigate programming languages whose types are indexed by algebraic structures such as groups of geometric transformations. Other examples include types indexed by principals–for information flow security–and types indexed by distances–for analysis of analytic uniform continuity properties. Following Reynolds, we prove a general Abstraction Theorem that covers all these instances. Consequences of our Abstraction Theorem include free theorems expressing invariance properties of programs, type isomorphisms based on invariance properties, and non-definability results indicating when certain algebraically indexed types are uninhabited or only inhabited by trivial programs. We have fully formalised our framework and most examples in Coq.

Categories and Subject Descriptors D.1.1 [*Programming techniques*]: Applicative (functional) programming; D.2.4 [*Software Engineering*]: Software/Program Verification; D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures

General Terms Languages, Theory, Types

Keywords parametricity, units of measure, dimensional analysis, invariance, computational geometry, information flow, metric types, uniform continuity

1. Introduction

The best way we know of describing the semantics of parametric polymorphism is *relational parametricity*, whose central result is Reynolds' Abstraction Theorem [18]. Its striking consequences include the well-known "free theorems" for polymorphic types [22], non-inhabitation results, and precise correspondences between System F encodings and algebraic datatypes [16], abstract data types, and, most recently, higher-order encodings of binder syntax [2].

Relational parametricity is in essence a principle of *invariance*: the behaviour of polymorphic code is invariant under changes of data representation. For example, the type $\forall\alpha.\texttt{list}\langle\alpha\rangle \to \texttt{list}\langle\alpha\rangle$ tells us that any transformation applied to elements of the input list will be reflected by the same transformation applied to elements of the result. Invariance results also abound in mathematics and physics. The area of a triangle is invariant with respect to isometries of the Euclidean plane; the determinant of a matrix is invariant under changes of basis; and Newton's laws are the same in all inertial frames. Typically, the transformations under which invariants are preserved have interesting structure: for example, translations in the Euclidean plane form an abelian group.

Inspired by this connection, we study type systems that capture rich invariants in types indexed by attributes with algebraic structure. For example, in computational geometry, points in the plane can be indexed by attributes representing affine transformations; in information-flow security, computations can be indexed by principals; in differential privacy, types can be indexed by 'distance'. Types that are polymorphic over such indices induce invariance properties and abstraction barriers beyond those introduced by their unindexed versions, as we shall illustrate. This generalises previous work by the third author on types parameterized by units of measure, whose invariance properties relate to changes of units, or *scaling* [13].

Invariance To illustrate type-induced invariance properties, consider two-dimensional geometry. In a conventional type system, a function $\mathrm{areaTri}$ that computes the area of a triangle might be assigned the type: $\texttt{vec} \times \texttt{vec} \times \texttt{vec} \to \texttt{real}$. But in our proposed system we can assign it the following more expressive polymorphic type:

$$\mathrm{areaTri} : \forall t{:}\mathsf{T}_2.\texttt{vec}\langle t\rangle \times \texttt{vec}\langle t\rangle \times \texttt{vec}\langle t\rangle \to \texttt{real}$$

This type expresses the fact that if each of the arguments to $\mathrm{areaTri}$ is translated by the same vector, then the result remains the same, that is, it is *invariant* under translation. Formally, for any vector $\vec{t}$,

$$\mathrm{areaTri}\ (\vec{t}+\vec{v_1}, \vec{t}+\vec{v_2}, \vec{t}+\vec{v_3}) = \mathrm{areaTri}\ (\vec{v_1}, \vec{v_2}, \vec{v_3})$$

Transformations typically *compose* in various ways, and the compositions satisfy algebraic laws. For example, we can assign a function that computes the area of a circle given its radius the following polymorphic type:

$$\mathrm{areaCircle} : \forall s{:}\mathsf{GL}_1.\texttt{real}\langle s\rangle \to \texttt{real}\langle s \cdot s\rangle$$

This captures the fact that the area of a circle varies as the square of its radius, i.e., $\mathrm{areaCircle}(kr) = k^2 \cdot \mathrm{areaCircle}(r)$ for any $k \neq 0$ (the 'sorts' T_2 and GL_1 will be explained later). Here, s can be interpreted as the *units of measure* of the argument to $\mathrm{areaCircle}$, and '$\cdot$' composes units using the product. We can also add an

POPL'13, January 23–25, 2013, Rome, Italy.

inverse operation and identity unit of measure 1, and then impose the algebraic laws of abelian groups. This permits identification of, for example, $\texttt{real}\langle s \cdot s^{-1} \rangle$ with the type $\texttt{real}\langle 1 \rangle$ of dimensionless constants.

Abstraction In his original paper on parametricity, Reynolds asserted that *type structure is a syntactic discipline for enforcing levels of abstraction*. We see something analogous here: if all primitive operations are given types that reflect their behaviour under translation, then there is no way to 'break' this property. For example, there is no way that areaTri can depend on the actual coordinates of its inputs. Furthermore, the distinction between points and vectors that is often enforced through abstract data types [7] is captured here by indices instead. For example, the operation that takes two points and computes their vector difference can be assigned the type $\forall t{:}\mathsf{T}_2.\texttt{vec}\langle t \rangle \times \texttt{vec}\langle t \rangle \to \texttt{vec}\langle 0 \rangle$, reflecting the invariance of the result (a pure vector) under translations of the point arguments. As a result through types alone we can, in essence, derive so-called *coordinate-free* geometry [15].

The invariance properties discussed above can be seen as "free theorems" [22], but the abstraction afforded by polymorphic indexed types can also induce interesting type *isomorphisms*. The type of areaCircle above is in fact isomorphic to $\texttt{real}\langle 1 \rangle$. A moment's thought reveals why: what possible unary functions can be constructed whose outputs scale as the square of the scaling of their inputs? Answer: just those functions of the form $\lambda x.kx^2$ for some constant k. In this case, of course, we expect that $k = \pi$.

Relational parametricity To derive such invariance and abstraction properties of types, we adopt the techniques of relational parametricity. Over an underlying index-erasure semantics we construct binary relations parameterised by an environment ρ that describes how values of primitive type are related according to their indices. For example, values v and w of type $\texttt{real}\langle s \rangle$ are related when v "scales to" w according to an interpretation of s (i.e., $w = \rho(s) \cdot v$). Values of polymorphic type are related exactly when they are related for all possible interpretations of the quantified variable. For example, values v and w of type $\forall t{:}\mathsf{T}_2.\texttt{vec}\langle t \rangle \to \texttt{vec}\langle t \rangle$ are related when they are related at type $\texttt{vec}\langle t \rangle \to \texttt{vec}\langle t \rangle$ for all translations $\vec{t} \in \mathrm{T}_2$ associated with t.

As it happens, the relational interpretations given above are functional, relating one value uniquely to another. Other applications make use of primitive relations that are not simple functions. For example, in a type system in which the index in $\texttt{real}\langle s \rangle$ is interpreted not as a unit of measure, but as a measure of *closeness*, two values x and y of this type are related if $|x - y| < \rho(s)$ for a positive real number $\rho(s)$. Rather beautifully, the standard notion of uniform continuity can then be expressed as $\forall \epsilon{:}\mathsf{R}^{>0}.\ \exists \delta{:}\mathsf{R}^{>0}.\ \texttt{real}\langle \delta \rangle \to \texttt{real}\langle \epsilon \rangle$.

Motivations Our motivations for studying algebraically indexed types are threefold. First, we believe that, as with units of measure [14], practical programming language extensions will follow. For example, in computational geometry and graphics, attributes on points, vectors, and other geometric types could be used to prevent the mixing of different coordinate systems, or 'frames'. Second, type-based static analyses can be based on indexed types, for example, in effect systems [4], and, more speculatively, in continuity analysis [8]. Finally, we believe that expressing algebraic invariants through types has the potential to offer slick proof techniques for mechanized mathematics. Harrison has applied the invariance properties of geometric primitives to create elegant proofs in geometry, based on 'without loss of generality' principles [10]. The invariance properties are expressed and propagated using ad-hoc tactics; our types offer a more principled means of achieving the same end, and the 'wlog' principle itself is expressed through type isomorphisms.

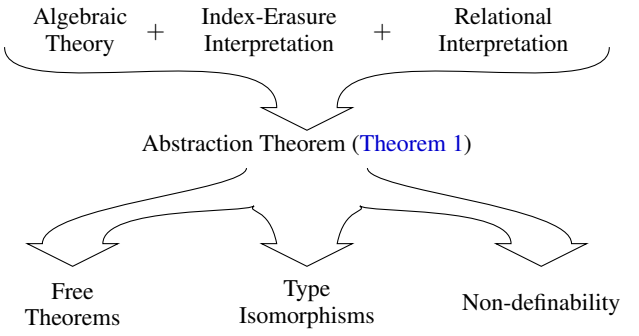

Figure 1. Summary of the Paper

We follow the mantra *semantics first, syntax later* in studying types with algebraic structure. We have not yet built a practical programming language that supports algebraically indexed types; nor have we designed type checking, type inference, or static analysis algorithms. But when we do so, the semantics will guide us. The fact that zero is polymorphic in units of measure (it can be given type $\forall u.\texttt{real}\langle u \rangle$) whereas other constants are dimensionless (having type $\texttt{real}\langle 1 \rangle$) is justified by the invariance properties induced by the types: zero is invariant under scaling, other constants are not. For less trivial constants and operations, the appropriate types are not so apparent, as we shall see, but invariance properties expressed by the semantics guide us in assigning appropriate types. Semantics does not lie!

1.1 Contributions

This paper makes the following specific contributions:

- We present a collection of compelling examples of algebraically indexed types, including a novel type system for geometry, a refined type system for information flow based on logic, and a simple type system with distance-indexed types.
- We formulate a type system that can either be used as a programming language in its own right, or as the target of type-based analyses. The type system consists of the usual type constructors together with a collection of indexed primitive types, universal and existential quantification over the indices, and a multi-sorted equational theory for indices.
- We describe a relational semantics for the type system and prove an analogue of Reynolds' Abstraction Theorem, for a given *model* of index sorts and relational interpretation of primitive types. We prove that the semantics soundly approximates contextual equivalence.
- For each of our main examples we deduce free theorems that are consequences of our Abstraction Theorem, prove specific non-definability results, and derive interesting type isomorphisms. For a large class of first-order types we give a general method for constructing suitable models to prove non-definability results. Figure 1 illustrates the central position of our analogue of Reynolds' Abstraction Theorem (Theorem 1) in these results.

We improve on the earlier semantics of units of measure [13] in a number of ways. By extending the language of units with an 'absolute value' operation, we can give more precise types and obtain more general invariance properties. The relational interpretation for units is both simpler and more flexible, and we derive slicker proofs of non-definability, and new results. Our notion of type isomorphism is stronger than before, being based on contextual equivalence.

We have fully formalised our framework and most examples in Coq, using strongly-typed term representations throughout [3]. The formalisation is available from
https://github.com/bobatkey/algebraically-indexed-types

1.2 Structure of paper

Our paper is structured as follows. In Section 2 we present an extended case study of two-dimensional geometry, with semi-formal description of types and results. In Section 3 we describe fully formally a general framework for algebraically-indexed types, and prove the Abstraction Theorem and soundness for semantic equivalence. Section 4 presents a number of applications of the theory to 2-d geometry, including free theorems, type isomorphisms and non-definability. Section 5 develops a more general technique for proving non-definability results. Section 6 presents the application of algebraically indexed types to information flow security, and Section 7 applies it to types indexed by distances. Finally, Section 8 discusses related work and future plans.

2. Geometry via Algebraically Indexed Types

We motivate our investigation of algebraically indexed types and their relational interpretations by developing a novel type system for programs that manipulate two-dimensional geometric data. Geometry is rich with operations that are invariant under transformation: affine operations are invariant under change of origin (Section 2.3), vector space operations are invariant under change of basis, and dot product is invariant under orthogonal changes of basis (Section 2.4). On the other hand, some geometric operations are interestingly variant under transformation. For example, cross products vary with scalings of the plane (Section 2.5). We incorporate (in)variance information about geometric primitives into type systems via algebraically indexed types.

2.1 Origin Invariance and Representation Independence

The basic data structure used in programs that manipulate geometric data is the n-tuple of numbers. In the 2-dimensional case, tuples $\vec{v} = (x, y)$ serve double duty, representing both *points*—offsets from some origin—and *vectors*—offsets in their own right. Despite their common representation, points and vectors are very different, and distinguishing between them is the key feature of *affine geometry* (see, for example, Chapter 2 of Gallier's book [9]). Nevertheless, computational geometry libraries traditionally either leave it to the programmer to maintain the distinction between points and vectors, or else use different abstract types for points and vectors to enforce it. In this paper we investigate a more sophisticated approach based on types indexed by change of origin transformations.

This approach regards the difference between points and vectors as a change of data representation. For example, if $(0, 0)$ and $(10, 20)$ are two origins, then the tuple $(1, 1)$ with respect to $(0, 0)$ and the tuple $(11, 21)$ with respect to $(10, 20)$ represent *the same point* because they have the same displacement from these two origins, respectively. This suggests that programs that manipulate points should be invariant with respect to changes of origin. Programs that manipulate vectors, on the other hand, should not be invariant under change of origin. Different vectors represent different offsets, and the vector $(0, 0)$ always represents the zero offset.

Invariance under change of representation immediately recalls Reynolds' fable about two professors teaching the theory of complex numbers [18]. One professor represents complex numbers using rectangular coordinates ($x+iy$), while the other represents them using polar coordinates ($\alpha\cos\theta + i\alpha\sin\theta$). Happily, after learning the basic operations on complex numbers in the two representations, the two classes can interact because the theory of complex numbers is invariant under the choice of representation. Reynolds formalises the idea of invariance under changes of representation as preservation of relations. For example, if a binary relation R relates the rectangular and polar representations of complex numbers, then a program that manipulates complex numbers at a level of abstraction above their specific representation should preserve R.

Reynolds' relational approach can be applied in the geometric setting to show how quantifying over all changes of origin ensures the invariance of programs under any particular choice of origin. For this, we first define a family of binary relations on $\mathbb{R}^2$ that is indexed by changes of origin. Changes of origin are represented by vectors in $\mathbb{R}^2$, and form a group T_2 of translations under addition. The T_2-indexed family of binary relations $\{R_{\vec{t}} \subseteq \mathbb{R}^2 \times \mathbb{R}^2\}_{\vec{t}\in\mathrm{T}_2}$ is then defined by $R_{\vec{t}} = \{(\vec{v}, \vec{v'}) \mid \vec{v'} = \vec{v} + \vec{t}\}$. We then consider a function f that takes as input two tuples in $\mathbb{R}^2$ and returns a single tuple in $\mathbb{R}^2$. We intend that the tuples all represent points with respect to the same origin, and that f is invariant under the choice of origin. Reynolds' relational approach formalises this intention precisely. For any $\vec{t} \in \mathrm{T}_2$:

$$\forall(\vec{v_1}, \vec{v_1'}) \in R_{\vec{t}}, (\vec{v_2}, \vec{v_2'}) \in R_{\vec{t}}.(f(\vec{v_1}, \vec{v_2}), f(\vec{v_1'}, \vec{v_2'})) \in R_{\vec{t}} \quad (1)$$

Unfolding the definition of $R_{\vec{t}}$ gives the equivalent formulation, again for all $\vec{t} \in \mathrm{T}_2$:

$$\forall\vec{v_1}, \vec{v_2}.\ f(\vec{v_1} + \vec{t}, \vec{v_2} + \vec{t}) = f(\vec{v_1}, \vec{v_2}) + \vec{t}. \quad (2)$$

Thus, Reynolds' preservation of relations, when instantiated with the family of relations $\{R_{\vec{t}}\}$, yields exactly the geometric property of invariance under change of origin.

2.2 A Type System for Change of Origin Invariance

Reynolds also showed how a type discipline can be used to establish that (the denotational interpretations of) programs preserve relations. For Reynolds, the type discipline of interest was that of the polymorphic λ-calculus, which supports the construction of new types by universal quantification over types. In terms of relations, Reynolds interprets universal quantification over types as quantification over binary relations between denotations of types. By contrast, in our statements of geometric invariance in Section 2.1 we did not quantify over all relations, but instead quantified over all changes of origin and used a specific choice of origin to select a relation from the family $\{R_{\vec{t}}\}$. This suggests introducing quantification over changes of origin into the language of types. We use the notation $\forall t{:}\mathsf{T}_2.A$ for quantification over all 2-dimensional translations (i.e., choices of origin) t, and refer to T_2 as the *sort* of t. Note the difference in fonts used to distinguish the semantic group T_2 from the syntactic sort T_2. We use a similar convention below, too.

Since the sort T_2 represents an abelian group, we can combine its elements using the usual group operations. We write operations additively, using $e_1 + e_2$ for the group operation, $-e$ for inverse and 0 for the unit. We also regard expressions built from variables and the group operations up to the abelian group axioms. For example, we regard $e_1 + (e_2 + e_3)$ and $(e_1 + e_2) + e_3$ as equivalent.

Our language of types includes the unit type `unit` and, for all types A and B, the function type $A \to B$, the sum type $A + B$, and the tuple type $A \times B$. We also assume a primitive type `real`, used to represent scalars. Although tuples of real numbers represent points and vectors in geometric applications, we cannot express this via the type `real` $\times$ `real`. Indeed, two elements of type `real` are related if and only if they are equal and, by Reynolds' interpretation of tuple types, two elements of `real` $\times$ `real` are also related if and only if they are equal. But since this does not give the correct relational interpretations for points and vectors, we introduce a new type $\texttt{vec}\langle e\rangle$, indexed by expressions e of sort T_2, to represent them. The index e represents the displacement by change of origin of a point of this type. Although we have taken pains to distinguish geometric points and vectors, we use the name `vec` for both to

recall the computer science notion of vector as a homogeneous sequence of values with a known length (in this case, 2).

As is standard for parametricity, every type has two interpretations: an index-erasure interpretation that ignores the indexing expression, and a relational interpretation as a binary relation on the index-erasure interpretation. We denote the index-erasure and relational interpretations with the notations $\lfloor\cdot\rfloor$ and $[\![\cdot]\!]$ respectively. To give such interpretations for the types $\mathtt{vec}\langle e\rangle$ and $\forall t{:}\mathsf{T}_2.A$, we assume for now that we can map each expression e of sort T_2 to an element $[\![e]\!]\rho$ of the group T_2 using some environment ρ that interprets e's free variables. The index-erasure and relational interpretations of $\mathtt{vec}\langle e\rangle$ are:

$$\begin{array}{lcl}\lfloor\mathtt{vec}\langle e\rangle\rfloor & = & \mathbb{R}^2 \\ [\![\mathtt{vec}\langle e\rangle]\!]\rho & = & R_{[\![e]\!]\rho} = \{(\vec{v},\vec{v'}) \mid \vec{v'} = \vec{v} + [\![e]\!]\rho\}\end{array}$$

The index-erasure and relational interpretations of $\forall t{:}\mathsf{T}_2.A$ are:

$$\begin{array}{lcl}\lfloor\forall t{:}\mathsf{T}_2.A\rfloor & = & \lfloor A\rfloor \\ [\![\forall t{:}\mathsf{T}_2.A]\!]\rho & = & \bigcap\{[\![A]\!](\rho,\vec{t}) \mid \vec{t}\in \mathrm{T}_2\}\end{array}$$

The index-erasure and relational interpretations are given formally in Sections 3.3 and 3.4.

At the end of Section 2.1 we considered functions $f : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}^2$ that preserve all changes of origin. This property of f can be expressed in terms of types by $f : \forall t{:}\mathsf{T}_2.\ \mathtt{vec}\langle t\rangle \times \mathtt{vec}\langle t\rangle \to \mathtt{vec}\langle t\rangle$. Spelling out the relational interpretation of this type using the definitions above and the standard relational interpretations for tuple and function types, we recover Statement 2 exactly.

2.3 Affine and Vector Operations

Invariance under change of origin is the key feature of affine geometry, whose central operation is the affine combination of points: $\lambda_1\vec{v_1} + \lambda_2\vec{v_2}$, where $\lambda_1 + \lambda_2 = 1$. Geometrically, this can be interpreted as describing all the points on the unique line through the points represented by $\vec{v_1}$ and $\vec{v_2}$ (assuming $\vec{v_1} \neq \vec{v_2}$). We add affine combination of points to our calculus as follows:

$$\begin{array}{l}\mathrm{affComb} : \forall t{:}\mathsf{T}_2.\ \mathtt{vec}\langle t\rangle \to \mathtt{real} \to \mathtt{vec}\langle t\rangle \to \mathtt{vec}\langle t\rangle \\ \lfloor\mathrm{affComb}\rfloor\ \vec{v_1}\ r\ \vec{v_2} = (1-r)\vec{v_1} + r\vec{v_2}\end{array}$$

It can be verified by hand that the index-erasure interpretation $\lfloor\mathrm{affComb}\rfloor$ is invariant under all changes of origin, as dictated by its type.

Example 1. The evaluation of quadratic Bézier curves (Bézier curves with two endpoints and a single control point) can be expressed using the affine combination primitive as follows:

$$\begin{array}{l}\mathrm{quadBézier} : \forall t{:}\mathsf{T}_2.\ \mathtt{vec}\langle t\rangle \to \mathtt{vec}\langle t\rangle \to \mathtt{vec}\langle t\rangle \to \mathtt{real} \to \mathtt{vec}\langle t\rangle \\ \mathrm{quadBézier}\ [t]\ p_0\ p_1\ p_2\ s = \\ \quad \mathrm{affComb}\ [t]\ (\mathrm{affComb}\ [t]\ p_0\ s\ p_1)\ s\ (\mathrm{affComb}\ [t]\ p_1\ s\ p_2)\end{array}$$

For two endpoints p_0 and p_2, a control point p_1, and $s \in [0,1]$, an application $\mathrm{quadBézier}\ p_0\ p_1\ p_2\ s$ gives the point on the curve at "time" s. The type of quadBézier immediately tells us that it preserves all changes of origin.

The obvious type for vector addition is $(+) : \mathtt{vec}\langle 0\rangle \to \mathtt{vec}\langle 0\rangle \to \mathtt{vec}\langle 0\rangle$. But we can reflect the fact that $(+)$ is not invariant under change of origin by giving it a more precise type that reflects how it varies with change of origin:

$$(+) : \forall t_1, t_2{:}\mathsf{T}_2.\ \mathtt{vec}\langle t_1\rangle \to \mathtt{vec}\langle t_2\rangle \to \mathtt{vec}\langle t_1 + t_2\rangle$$

Intuitively, this type says that if the first input vector has been displaced by t_1 and the second by t_2, then their sum is displaced by $t_1 + t_2$. We can also negate vectors, yielding a vector which points in the opposite direction. Negation negates translation arguments:

$$\mathrm{negate} : \forall t{:}\mathsf{T}_2.\ \mathtt{vec}\langle t\rangle \to \mathtt{vec}\langle -t\rangle$$

Finally, with the primitive operations of addition and negation of vectors we can define the derived operation of subtraction:

$$\begin{array}{l}(-) : \forall t_1, t_2{:}\mathsf{T}_2.\ \mathtt{vec}\langle t_1\rangle \to \mathtt{vec}\langle t_2\rangle \to \mathtt{vec}\langle t_1 - t_2\rangle \\ (-)\ [t_1]\ [t_2]\ p_1\ p_2 = p_1 + \mathrm{negate}\ p_2\end{array}$$

Given two points that are invariant with respect to the same change of origin—i.e., two values of type $\mathtt{vec}\langle t\rangle$—we can use subtraction to compute their offset:

$$\begin{array}{l}\mathrm{offset} : \forall t{:}\mathsf{T}_2.\ \mathtt{vec}\langle t\rangle \to \mathtt{vec}\langle t\rangle \to \mathtt{vec}\langle 0\rangle \\ \mathrm{offset}\ [t]\ p_1\ p_2 = p_1 - p_2\end{array}$$

The result is a vector expressed with respect to the null change of origin: note how the algebraic structure on the indexing theory induces type equalities that can be used to simplify the type of the result of offset from $\mathtt{vec}\langle t-t\rangle$ to $\mathtt{vec}\langle 0\rangle$. The type of $(+)$ can also be specialised to the case of moving a point by a vector:

$$\begin{array}{l}\mathrm{moveBy} : \forall t{:}\mathsf{T}_2.\ \mathtt{vec}\langle t\rangle \to \mathtt{vec}\langle 0\rangle \to \mathtt{vec}\langle t\rangle \\ \mathrm{moveBy}\ [t]\ p\ v = p + v\end{array}$$

The types we assign to the remaining vector space primitives, namely $0 : \mathtt{vec}\langle 0\rangle$ for the zero vector and $(*) : \mathtt{real} \to \mathtt{vec}\langle 0\rangle \to \mathtt{vec}\langle 0\rangle$ for multiplication by a scalar, do not describe any interesting effects on translations.

Example 2. The vector space operators and the properties that follow from their types allow us to establish a useful type isomorphism. Consider functions with types following the schema:

$$\tau_n \stackrel{def}{=} \forall t{:}\mathsf{T}_2.\ \underbrace{\mathtt{vec}\langle t\rangle \to \ldots \to \mathtt{vec}\langle t\rangle}_{n+1 \text{ times}} \to \mathtt{real}$$

Just by looking at the types τ_n, we know that their inhabitants will be invariant under change of origin because of the quantification over all t in T_2. So we may as well choose one of the input points as the origin and assume that all the other points are defined with respect to it. This formalises the common mathematical practice of stating that "without loss of generality" we can take some point in a description of a problem to be the origin provided the problem statement is invariant under translation. Each type τ_n is isomorphic to the corresponding type σ_n:

$$\sigma_n \stackrel{def}{=} \underbrace{\mathtt{vec}\langle 0\rangle \to \ldots \mathtt{vec}\langle 0\rangle}_{n \text{ times}} \to \mathtt{real}$$

We demonstrate these isomorphisms formally in Section 4, in the more general setting of types indexed by abelian groups.

Example 3. So far we have emphasised the derivation of properties, or "free theorems", of programs from their types. But using more refined relational interpretations of types we can also show that certain types are uninhabited. For example, the type $\forall t{:}\mathsf{T}_2.\ \mathtt{vec}\langle t + t\rangle \to \mathtt{vec}\langle t\rangle$ has no inhabitants. Intuitively, this is because we cannot remove the extra t in $\mathtt{vec}\langle t + t\rangle$ using the vector operations. We formalise this non-definability result in Section 5 using a specialised relational interpretation.

2.4 Change of Basis Invariance

Although vector addition, negation, and scaling are not invariant under change of origin, they are invariant under change of basis. As with origin invariance, we can express basis invariance as preservation of relations indexed by changes of basis. Change of basis is achieved by applying an invertible linear map, and the collection of all such maps on $\mathbb{R}^2$ forms the *General Linear* group GL_2, which we represent in our language by a new indexing sort GL_2 with (non-abelian) group structure that we will write multiplicatively. We then extend $\mathtt{vec}$ to allow indices of sort GL_2, as well as T_2, so that $\mathtt{vec}\langle B, t\rangle$ is a vector that varies with change of basis B

$$
\begin{array}{rcl}
0 & : & \forall s{:}\mathsf{GL}_1.\ \mathtt{real}\langle s\rangle \\
1 & : & \mathtt{real}\langle 1\rangle \\
(+) & : & \forall s{:}\mathsf{GL}_1.\ \mathtt{real}\langle s\rangle \to \mathtt{real}\langle s\rangle \to \mathtt{real}\langle s\rangle \\
(-) & : & \forall s{:}\mathsf{GL}_1.\ \mathtt{real}\langle s\rangle \to \mathtt{real}\langle s\rangle \to \mathtt{real}\langle s\rangle \\
(*) & : & \forall s_1, s_2{:}\mathsf{GL}_1.\ \mathtt{real}\langle s_1\rangle \to \mathtt{real}\langle s_2\rangle \to \mathtt{real}\langle s_1 s_2\rangle \\
(/) & : & \forall s_1, s_2{:}\mathsf{GL}_1.\ \mathtt{real}\langle s_1\rangle \to \mathtt{real}\langle s_2\rangle \\
 & & \qquad\qquad \to \mathtt{real}\langle s_1 s_2^{-1}\rangle + \mathtt{unit} \\
\mathrm{abs} & : & \forall s{:}\mathsf{GL}_1.\ \mathtt{real}\langle s\rangle \to \mathtt{real}\langle |s|\rangle
\end{array}
$$

Figure 2. Operations on scaled real numbers

and change of origin t. Formally, the index-erasure and relational semantics of $\mathtt{vec}\langle B, t\rangle$ are given by:

$$
\begin{array}{rcl}
\lfloor \mathtt{vec}\langle e_B, e_t\rangle \rfloor & = & \mathbb{R}^2 \\
[\![\mathtt{vec}\langle e_B, e_t\rangle]\!]\rho & = & \{(\vec{v}, \vec{v'}) \mid \vec{v'} = ([\![e_B]\!]\rho)\vec{v} + [\![e_t]\!]\rho\}
\end{array}
$$

Affine Geometry An *affine transformation* is an invertible linear map together with a translation. We can assign types to all the primitive affine and vector space operations indicating how they they behave with respect to affine transformations:

$$
\begin{array}{rcl}
\mathrm{affComb} & : & \forall B{:}\mathsf{GL}_2, t{:}\mathsf{T}_2. \\
 & & \quad \mathtt{vec}\langle B, t\rangle \to \mathtt{real} \to \mathtt{vec}\langle B, t\rangle \to \mathtt{vec}\langle B, t\rangle \\
(+) & : & \forall B{:}\mathsf{GL}_2, t_1, t_2{:}\mathsf{T}_2. \\
 & & \quad \mathtt{vec}\langle B, t_1\rangle \to \mathtt{vec}\langle B, t_2\rangle \to \mathtt{vec}\langle B, t_1 + t_2\rangle \\
\mathrm{negate} & : & \forall B{:}\mathsf{GL}_2, t{:}\mathsf{T}_2.\ \mathtt{vec}\langle B, t\rangle \to \mathtt{vec}\langle B, -t\rangle \\
0 & : & \forall B{:}\mathsf{GL}_2.\ \mathtt{vec}\langle B, 0\rangle \\
(*) & : & \forall B{:}\mathsf{GL}_2.\ \mathtt{real} \to \mathtt{vec}\langle B, 0\rangle \to \mathtt{vec}\langle B, 0\rangle
\end{array}
$$

Euclidean Geometry Euclidean geometry extends affine geometry with the *dot product*, or *inner product*, operation of two vectors. The dot product is defined by $(x_1, y_1)\cdot(x_2, y_2) = x_1x_2 + y_1y_2$. To assign it a type we note that, although dot product is not invariant under GL_2 or T_2, it is invariant under the subgroup O_2 of GL_2 of *orthogonal* linear transformations, i.e., the subgroup of invertible linear maps whose matrix representations' transposes are equal to their inverses. We thus introduce a new sort O_2 of orthogonal transformations, and overload the multiplicative group operations for inhabitants of O_2. Further assuming an injection ι_O that takes $e : \mathsf{O}_2$ to $\iota_O(e) : \mathsf{GL}_2$ we assign dot product this type:

$$(\cdot) : \forall O{:}\mathsf{O}_2.\ \mathtt{vec}\langle \iota_O O, 0\rangle \to \mathtt{vec}\langle \iota_O O, 0\rangle \to \mathtt{real}$$

The cross product of two vectors is defined on coordinate representations as $(x_1, y_1) \times (x_2, y_2) = x_1y_2 - x_2y_1$. Geometrically, the cross product is the signed area of the parallelogram described by the pair of input vectors. Under change of basis by an invertible linear transformation B, the cross product of two vectors varies with the determinant of B. This corresponds to scaling the plane by the change of basis transformation, so we augment our calculus with a new sort GL_1 of scale factors (i.e., 1-dimensional invertible linear maps). Semantically, GL_1 ranges over the non-zero real numbers and forms an abelian group which we write multiplicatively. We also add two new operations: determinant, $\det B$, which takes an inhabitant of GL_2 to its determinant in GL_1, and absolute value, $|e|$, which takes scaling factors to scaling factors. We also refine the type $\mathtt{real}$ of real numbers so that it is indexed by the sort GL_1: $\mathtt{real}\langle e\rangle$. The old type $\mathtt{real}$ is then just $\mathtt{real}\langle 1\rangle$, and the full collection of operations on real numbers indexed by scaling factors is shown in Figure 2. We can thus assign cross product the type:

$$(\times) : \forall B{:}\mathsf{GL}_2.\ \mathtt{vec}\langle B, 0\rangle \to \mathtt{vec}\langle B, 0\rangle \to \mathtt{real}\langle \det B\rangle$$

Since the absolute value of the determinant of an orthogonal transformation is always 1, we assume $|\det(\iota_O O)| = 1$ to hold for any $O \in \mathsf{O}_2$.

Example 4. We can use the operations of this subsection to compute the area of a triangle. We have:

$$
\begin{array}{l}
\mathrm{area} : \forall B{:}\mathsf{GL}_2, t{:}\mathsf{T}_2. \\
\qquad \mathtt{vec}\langle B, t\rangle \to \mathtt{vec}\langle B, t\rangle \to \mathtt{vec}\langle B, t\rangle \to \mathtt{real}\langle |\det B|\rangle \\
\mathrm{area}\ [B]\ [t]\ p_1\ p_2\ p_3 = \frac{1}{2} * \mathrm{abs}\ ((p_2 - p_1) \times (p_3 - p_1))
\end{array}
$$

The calculation is performed in several steps, each of which removes some of the symmetry described by the type of area. First, the two offset vectors $p_2 - p_1$ and $p_3 - p_1$ are computed. These operations remove the effect of translations on the result in exactly the same way as the type isomorphism in Example 2. Next, we compute the cross product of the two vectors, which gives the area of the parallelogram described by the sides of the triangle and has type $\mathtt{real}\langle \det B\rangle$. This removes some of the symmetry due to invertible linear maps, but the cross product still varies with the sign of the determinant. We remove this symmetry as well using abs. This gives a value of type $\mathtt{real}\langle |\det B|\rangle$ which we multiply by $\frac{1}{2}$ to recover the area of the triangle rather than that of the whole parallelogram. If we specialise area to just orthogonal transformations, the assumption $|\det(\iota_O O)| = 1$ gives the following type:

$$
\begin{array}{l}
\mathrm{area} : \forall O{:}\mathsf{O}_2, t{:}\mathsf{T}_2. \\
\qquad \mathtt{vec}\langle \iota_O O, t\rangle \to \mathtt{vec}\langle \iota_O O, t\rangle \to \mathtt{vec}\langle \iota_O O, t\rangle \to \mathtt{real}\langle 1\rangle
\end{array}
$$

This type shows that the area of a triangle is invariant under orthogonal transformations and translations. Combinations of such transformations are *isometries*, i.e., distance preserving maps.

2.5 Scale Invariance and Dimensional Analysis

Indexing types by scaling factors brings us to the original inspiration for the current work: Kennedy's interpretation of his units of measure type system via scaling invariance [13]. Kennedy shows how interpreting types in terms of scaling invariance brings the techniques of dimensional analysis to bear on programming. The types of the real number arithmetic operations in Figure 2 are exactly the types Kennedy assigns in his units of measure system, except for that of the absolute value operation. Semantically, our type indexes by non-zero scaling factors, whereas Kennedy's indexes by strictly positive ones.

In our two-dimensional setting we can add to Kennedy's one-dimensional scaling invariance an operation ι_1 that, semantically, takes scale factors in GL_1 to invertible linear maps in GL_2, i.e., takes numbers s to matrices $\left(\begin{smallmatrix} s & 0 \\ 0 & s \end{smallmatrix}\right)$. This operation satisfies the equation $\det(\iota_1 s) = s^2$, indicating that scaling the plane by s in both directions scales areas by s^2.

Example 5. Just as we specialised the type of the area function to orthogonal transformations in Example 4, we can also specialise area's type to scaling transformations. This yields the type:

$$
\begin{array}{l}
\mathrm{area} : \forall s{:}\mathsf{GL}_1, t{:}\mathsf{T}_2. \\
\qquad \mathtt{vec}\langle \iota_1 s, t\rangle \to \mathtt{vec}\langle \iota_1 s, t\rangle \to \mathtt{vec}\langle \iota_1 s, t\rangle \to \mathtt{real}\langle s^2\rangle
\end{array}
$$

As expected, the area of a triangle varies with the square of scalings of the plane, and this is reflected in the type.

Linear maps of the form $\left(\begin{smallmatrix} s & 0 \\ 0 & s \end{smallmatrix}\right)$, as generated by ι_1, commute with all other invertible linear maps. We thus require $(\iota_1 s)B = B(\iota_1 s)$ to hold. The scaling maps $\left(\begin{smallmatrix} s & 0 \\ 0 & s \end{smallmatrix}\right)$ are precisely the elements of GL_2 that commute with all others; these form the *centre* of GL_2. If we keep track of scalings, then we can assign the more precise types to scalar multiplication and dot product. These are shown in Figure 3, which summarises the most general types of all the vector operations that we have described.

$$
\begin{aligned}
0 &: \forall B{:}\mathsf{GL}_2.\ \mathtt{vec}\langle B, 0\rangle \\
(+) &: \forall B{:}\mathsf{GL}_2, t_1, t_2{:}\mathsf{T}_2. \\
&\qquad \mathtt{vec}\langle B, t_1\rangle \to \mathtt{vec}\langle B, t_2\rangle \to \mathtt{vec}\langle B, t_1 + t_2\rangle \\
\text{negate} &: \forall B{:}\mathsf{GL}_2, t{:}\mathsf{T}_2.\ \mathtt{vec}\langle B, t\rangle \to \mathtt{vec}\langle B, -t\rangle \\
(*) &: \forall s{:}\mathsf{GL}_1, B{:}\mathsf{GL}_2. \\
&\qquad \mathtt{real}\langle s\rangle \to \mathtt{vec}\langle B, 0\rangle \to \mathtt{vec}\langle \iota_1(s)B, 0\rangle \\
\text{affComb} &: \forall B{:}\mathsf{GL}_2, t{:}\mathsf{T}_2. \\
&\qquad \mathtt{vec}\langle B, t\rangle \to \mathtt{real}\langle 1\rangle \to \mathtt{vec}\langle B, t\rangle \to \mathtt{vec}\langle B, t\rangle \\
(\cdot) &: \forall s{:}\mathsf{GL}_1, O{:}\mathsf{O}_2.\ \mathtt{vec}\langle \iota_1(s)\iota_O(O), 0\rangle \to \\
&\qquad\qquad \mathtt{vec}\langle \iota_1(s)\iota_O(O), 0\rangle \to \mathtt{real}\langle s^2\rangle \\
(\times) &: \forall B{:}\mathsf{GL}_2.\ \mathtt{vec}\langle B, 0\rangle \to \mathtt{vec}\langle B, 0\rangle \to \mathtt{real}\langle \det B\rangle
\end{aligned}
$$

Figure 3. Operations on vectors

Example 6. With the operations in Figure 2, it is not possible to write a term with the following type that is not constantly zero:

$$\forall s{:}\mathsf{GL}_2.\ \mathtt{real}\langle s^2\rangle \to \mathtt{real}\langle s\rangle$$

This was shown by Kennedy for his units of measure system [13]. In particular, it is not possible to write a square root function with the above type. The non-definability of square root is similar to the uninhabitation of the type in Example 3.

In Section 4.3 we revisit Kennedy's result and show that even if we add square root as a primitive operation—with the type above—then it is still not possible to construct the cube root function. The non-definability of cube root is related to the impossibility of trisecting an arbitrary angle by ruler and compass constructions.

3. A General Framework

We now present our framework for algebraically indexed types and its relational interpretation. We define the syntax of algebraically indexed types (Section 3.1) and a syntax for terms in a general programming language for algebraically indexed types (Section 3.2). We give an index-erasure semantics to types and terms (Section 3.3), and based on this semantics define notions of contextual equivalence and type isomorphism. We then introduce a relational semantics for types parameterised by an appropriate 'model' of the algebraic theory (Section 3.4), prove the central Abstraction Theorem and use the relational semantics to define a notion of semantic equivalence that soundly approximates contextual equivalence (Section 3.5).

We will use a type system for affine geometry as a running example throughout, so that by the end of the section we have prepared enough syntactic and semantic gadgets to let us prove invariance and abstraction properties for geometric examples in Section 4.

3.1 Algebraically-Indexed Types

The index expressions and types of an instantiation of our general framework are derived from the following data:

1. A collection Sort of index sorts. We use the meta-syntactic variables $s, s_1, s_2, \ldots$ for arbitrary sorts taken from Sort.
2. A collection $\mathit{IndexOp}$ of index operations, with a function $\mathrm{opArity} : \mathit{IndexOp} \to \mathit{Sort}^* \times \mathit{Sort}$. (We use the notation A^* to denote the set of lists of elements of some set A.)
3. A collection $\mathit{PrimType}$ of primitive types, with a function $\mathrm{tyArity} : \mathit{PrimType} \to \mathit{Sort}^*$, describing the sorts of the arguments of each primitive type.

Example (Geometry: syntax)**.** The two-dimensional geometry system has a sort for each of the geometric groups mentioned in Section 2, so $\mathit{Sort} = \{\mathsf{T}_2, \mathsf{GL}_2, \mathsf{O}_2, \mathsf{GL}_1\}$. We have additive group structure on T_2, multiplicative group structure on GL_1, GL_2, and O_2, injections from O_2 and GL_1 into GL_2, determinant, and absolute value. Thus, $\mathit{IndexOp} = \{0, +, -, 1_G, - \cdot_G -, -^{-1_G}, \iota_O, \iota_1, \det, |\cdot|\}$, where $G \in \{\mathsf{GL}_1, \mathsf{GL}_2, \mathsf{O}_2\}$, and

$$
\begin{array}{ll|ll}
\mathrm{opArity}(0) & =([], \mathsf{T}_2) & \mathrm{opArity}(1_G) & =([], G) \\
\mathrm{opArity}(+) & =([\mathsf{T}_2, \mathsf{T}_2], \mathsf{T}_2) & \mathrm{opArity}(\cdot_G) & =([G, G], G) \\
\mathrm{opArity}(-) & =([\mathsf{T}_2], \mathsf{T}_2) & \mathrm{opArity}(^{-1_G}) & =([G], G) \\
\mathrm{opArity}(\iota_O) & =([\mathsf{O}_2], \mathsf{GL}_2) & \mathrm{opArity}(\iota_1) & =([\mathsf{GL}_1], \mathsf{GL}_2) \\
\mathrm{opArity}(\det) & =([\mathsf{GL}_2], \mathsf{GL}_1) & \mathrm{opArity}(|\cdot|) & =([\mathsf{GL}_1], \mathsf{GL}_1)
\end{array}
$$

The intended interpretations of the top three pairs of operations are group unit, group combination and group negation, respectively. When we discuss equational theories on index expressions in Section 3.1.2 we will impose the (abelian) group laws. For this example, we also have $\mathit{PrimType} = \{\mathtt{vec}, \mathtt{real}\}$, with $\mathrm{tyArity}(\mathtt{vec}) = [\mathsf{GL}_2, \mathsf{T}_2]$ and $\mathrm{tyArity}(\mathtt{real}) = [\mathsf{GL}_1]$. □

We assume a countably infinite collection of index variable names i, i_1, i_2, *etc.* *Index contexts* $\Delta = i_1{:}s_1, \ldots, i_n{:}s_n$ are lists of variable/sort pairs such that all the variable names are distinct. The rules in Figure 4 generate two judgements: well-sorted index expressions $\Delta \vdash e : s$ and well-indexed types $\Delta \vdash A\ \mathsf{type}$. Since index variables may appear in types, types are judged to be well-indexed with respect to an index context Δ. The rules for well-sorted index expressions are particularly simple: either an index expression is a variable that appears in the context (rule IVar), or it is an application of an index operation taken from $\mathit{IndexOp}$ to other index expressions (rule IOp). The rules for well-indexed types include the usual ones for the simply-typed λ-calculus with unit, sum and tuple types (rules TyUnit, TyArr, TyTuple and TySum). We use `bool` as an abbreviation for `unit + unit`. The rule TyPrim forms, from a primitive type `X` and appropriately sorted index expressions $e_1, \ldots, e_n$, the well-indexed type $\mathtt{X}\langle e_1, \ldots, e_n\rangle$. The rule TyForall forms universally quantified types, where the universal quantification ranges over all index expressions of some sort. Existential types, formed using the TyEx rule, allow for abstraction by hiding.

3.1.1 Substitution of Index Expressions

It is convenient to express substitution of index expressions in terms of simultaneous substitutions. Given a pair of index contexts Δ and $\Delta' = i_1{:}s_1, \ldots, i_n{:}s_n$, a (simultaneous) *substitution* $\Delta \vdash \sigma \Rightarrow \Delta'$ is a sequence of expressions $\sigma = (e_1, \ldots, e_n)$ such that $\Delta \vdash e_j : s_j$ for all $1 \le j \le n$. Given a substitution $\Delta \vdash \sigma = (e_1, \ldots, e_n) \Rightarrow \Delta'$ and a variable $i_j{:}s_j$ in Δ', we write $\sigma(i_j)$ for the index expression e_j. We write $\Delta \Rightarrow \Delta'$ for the set of all substitutions σ such that $\Delta \vdash \sigma \Rightarrow \Delta'$. We can think of any sequence of sorts as an index context. In particular, we will make use of substitutions of the form $\Delta \vdash \sigma \Rightarrow \mathrm{tyArity}(\mathtt{X})$, since these are exactly sequences of index arguments suitable for the primitive type `X`. By further abuse of notation, we write $\Delta \Rightarrow \mathrm{tyArity}(\mathtt{X})$ for the set of all substitutions σ such that $\Delta \vdash \sigma \Rightarrow \mathrm{tyArity}(\mathtt{X})$.

For a substitution $\Delta \vdash \sigma \Rightarrow \Delta'$, where $\Delta' = i_1{:}s_1, \ldots, i_n{:}s_n$, and a variable/sort pair $i{:}s$ such that i does not appear in either Δ or Δ', we can form the *lifted* substitution $\Delta, i{:}s \vdash \sigma_{i:s} = (\sigma(i_1), \ldots, \sigma(i_n), i) \Rightarrow \Delta', i{:}s$. Application of a substitution $\Delta \vdash \sigma \Rightarrow \Delta'$ to a well-sorted index expression $\Delta' \vdash e : s$ yields a well-sorted index expression $\Delta \vdash \sigma^* e : s$. The expression $\sigma^* e$ is defined on variables as $\sigma^* i \stackrel{def}{=} \sigma(i)$, and on operation symbols as $\sigma^*(\mathtt{f}(e_1, \ldots, e_n)) \stackrel{def}{=} \mathtt{f}(\sigma^* e_1, \ldots, \sigma^* e_n)$. Given $\Delta' \vdash A\ \mathsf{type}$, we have $\Delta \vdash \sigma^* A\ \mathsf{type}$. The key clauses defining $\sigma^* A$ are for

Well-sorted index expressions

$$\frac{i : s \in \Delta}{\Delta \vdash i : s}\ \textsc{IVar} \qquad \frac{\mathtt{f} \in \mathit{IndexOp} \qquad \mathrm{opArity}(\mathtt{f}) = ([s_1, \ldots, s_n], s) \qquad \{\Delta \vdash e_j : s_j\}_{1 \le j \le n}}{\Delta \vdash \mathtt{f}(e_1, \ldots, e_n) : s}\ \textsc{IOp}$$

Well-indexed types

$$\frac{\mathtt{X} \in \mathit{PrimType} \qquad \mathrm{tyArity}(\mathtt{X}) = [s_1, \ldots, s_n] \qquad \{\Delta \vdash e_j : s_j\}_{1 \le j \le n}}{\Delta \vdash \mathtt{X}\langle e_1, \ldots, e_n\rangle\ \mathsf{type}}\ \textsc{TyPrim} \qquad \frac{}{\Delta \vdash \mathtt{unit}\ \mathsf{type}}\ \textsc{TyUnit} \qquad \frac{\Delta \vdash A\ \mathsf{type} \qquad \Delta \vdash B\ \mathsf{type}}{\Delta \vdash A \to B\ \mathsf{type}}\ \textsc{TyArr}$$

$$\frac{\Delta \vdash A\ \mathsf{type} \qquad \Delta \vdash B\ \mathsf{type}}{\Delta \vdash A \times B\ \mathsf{type}}\ \textsc{TyTuple} \qquad \frac{\Delta \vdash A\ \mathsf{type} \qquad \Delta \vdash B\ \mathsf{type}}{\Delta \vdash A + B\ \mathsf{type}}\ \textsc{TySum} \qquad \frac{\Delta, i{:}s \vdash A\ \mathsf{type}}{\Delta \vdash \forall i{:}s.A\ \mathsf{type}}\ \textsc{TyForall} \qquad \frac{\Delta, i{:}s \vdash A\ \mathsf{type}}{\Delta \vdash \exists i{:}s.A\ \mathsf{type}}\ \textsc{TyEx}$$

Figure 4. Index expressions and types

primitive types and the universal and existential quantifiers:

$$\sigma^*(\mathtt{X}\langle e_1, \ldots, e_n\rangle) \stackrel{def}{=} \mathtt{X}\langle \sigma^* e_1, \ldots, \sigma^* e_n\rangle$$
$$\sigma^*(\forall i{:}s.A) \stackrel{def}{=} \forall i{:}s.\sigma^*_{i:s} A \qquad \sigma^*(\exists i{:}s.A) \stackrel{def}{=} \exists i{:}s.\sigma^*_{i:s} A$$

The *identity* substitution $\Delta \vdash \mathrm{id}_\Delta \Rightarrow \Delta$ is $\mathrm{id}_\Delta = (i_1, \ldots, i_n)$ where $\Delta = i_1{:}s_1, \ldots, i_n{:}s_n$. The *composition* of two substitutions $\Delta \vdash \sigma \Rightarrow \Delta'$ and $\Delta' \vdash \sigma' \Rightarrow \Delta''$, where $\sigma' = (e'_1, \ldots, e'_n)$, is defined as $\Delta \vdash \sigma' \circ \sigma \stackrel{def}{=} (\sigma^* e'_1, \ldots, \sigma^* e'_n) \Rightarrow \Delta''$. Given a context $\Delta = i_1{:}s_1, \ldots, i_n{:}s_n$, and a variable/sort pair $i{:}s$ such that i does not appear in Δ, we define the *projection* substitution $\Delta, i{:}s \vdash \pi_{i:s} \Rightarrow \Delta$ as $\pi_{i:s} = (i_1, \ldots, i_n)$.

3.1.2 Index Expression Equality and Type Equality

Much of the power of indexing types by the expressions of an algebraic theory comes from the equations of the theory. For example, in Section 2 the types $\mathtt{vec}\langle B, t_1 + t_2\rangle$ and $\mathtt{vec}\langle B, t_2 + t_1\rangle$ are considered equal by the type system because $+$ is commutative. In the general framework, the equations between types are derived from a set $\mathit{IndexAx}$ of axioms $\Delta \vdash e \stackrel{ax}{\equiv} e' : s$ that are well-sorted, in the sense that both $\Delta \vdash e : s$ and $\Delta \vdash e' : s$ hold.

Given a set $\mathit{IndexAx}$ of axioms, we generate the equality judgment between index expressions $\Delta \vdash e \equiv e' : s$ by a set of rules. The following rule lets us use substitution instances of axioms:

$$\frac{(\Delta' \vdash e \stackrel{ax}{\equiv} e' : s) \in \mathit{IndexAx} \qquad \Delta \vdash \sigma \Rightarrow \Delta'}{\Delta \vdash \sigma^* e \equiv \sigma^* e' : s}$$

We also assume the standard congruence, reflexivity, symmetry and transitivity rules for the equality judgment.

Example (Geometry: axioms)**.** In Section 2 we assumed various equational axioms for indexing expressions standing for elements of geometric groups. Assuming the abelian group axioms for translations we can formalise this in our framework:

$$t : \mathsf{T}_2 \vdash t + 0 \stackrel{ax}{\equiv} t : \mathsf{T}_2$$
$$t_1, t_2, t_3 : \mathsf{T}_2 \vdash t_1 + (t_2 + t_3) \stackrel{ax}{\equiv} (t_1 + t_2) + t_3 : \mathsf{T}_2$$
$$t : \mathsf{T}_2 \vdash t + (-t) \stackrel{ax}{\equiv} 0 : \mathsf{T}_2$$
$$t_1, t_2 : \mathsf{T}_2 \vdash t_1 + t_2 \stackrel{ax}{\equiv} t_2 + t_1 : \mathsf{T}_2$$

Similarly, the sort of scale factors GL_1 forms an abelian group under multiplication, and the sorts GL_2 and O_2 form (non-abelian) multiplicative groups, so we assume the appropriate axioms. We also assume that the operations $\iota_O, \iota_1, \det$ and $|\cdot|$ are group homomorphisms, and that expressions of the form $\iota_1(s)$ commute with group multiplication in the sort GL_2. The absolute value of the determinant of an orthgonal transformation is always 1, so we also assume $|\det(\iota_O O)| \stackrel{ax}{\equiv} 1$. Similarly, scaling maps have a determinant expressible in terms of other operations: $\det(\iota_1(s)) \stackrel{ax}{\equiv} s \cdot s$. We also assume the axiom $|s^2| \stackrel{ax}{\equiv} s^2$. □

The equality judgment $\Delta \vdash e \equiv e' : s$ on index expressions generates the equality judgment $\Delta \vdash A \equiv B\ \mathsf{type}$ on types. The basic rule generating equality judgments on types equates applications of primitive types if their arguments are equal:

$$\frac{\{\Delta \vdash e_j \equiv e'_j : s_j\}_{1 \le j \le n}}{\Delta \vdash \mathtt{X}\langle e_1, \ldots, e_n\rangle \equiv \mathtt{X}\langle e'_1, \ldots, e'_n\rangle\ \mathsf{type}}$$

The rest of the rules for equality on types ensure that it is a congruence relation and an equivalence relation.

The substitutions $\Delta \vdash \sigma \Rightarrow \Delta'$ and $\Delta \vdash \sigma' \Rightarrow \Delta'$ are defined to be equal, and written $\Delta \vdash \sigma \equiv \sigma' \Rightarrow \Delta'$, if their component expressions are equal in the context Δ: i.e., if $\Delta \vdash e_j \equiv e'_j : s_j$, for all j.

3.2 Well-typed terms

We now present the rules for well-typed terms over the collection of types we defined in Section 3.1.

Well-typed terms are defined with respect to well-indexed typing contexts, which are in turn defined with respect to an index context. Well-indexed typing contexts with respect to an index context Δ are sequences of variable/type pairs with no repeated variable names such that each type is well-indexed with respect to Δ. Formally, well-indexed typing contexts are given by

$$\frac{}{\Delta \vdash \epsilon\ \mathsf{ctxt}} \qquad \frac{\Delta \vdash \Gamma\ \mathsf{ctxt} \qquad \Delta \vdash A\ \mathsf{type} \qquad x \notin \Gamma}{\Delta \vdash \Gamma, x : A\ \mathsf{ctxt}}$$

Application of substitutions extends to typing contexts by applying the substitution to each type.

Well-typed terms are defined with respect to an index context Δ and a type context $\Delta \vdash \Gamma\ \mathsf{ctxt}$. The judgment $\Delta; \Gamma \vdash M : A$ is defined in Figure 5. The equational theory on types is incorporated into the type system via the rule TyEq, which allows a term that has type A to also have any equal type B as well.

For any particular theory we assume that there is a closed typing context Γ_{ops} that describes the types of the primitive operations.

Example (Geometry: operations)**.** For geometry Γ_{ops} would collect together the types of primitive operations as listed in Figure 2 and Figure 3. □

3.3 Index-Erasure Semantics

Having defined the syntax of algebraically indexed types and terms, we turn to their denotational interpretation. We first define an *index-erasure* interpretation of types and terms that interprets every well-indexed type as a set, ignoring the indexing expressions, and which

$$\frac{\Delta \vdash \Gamma\ \mathsf{ctxt} \qquad x : A \in \Gamma}{\Delta; \Gamma \vdash x : A}\ \textsc{Var} \qquad \frac{\Delta; \Gamma \vdash M : A \qquad \Delta \vdash A \equiv B\ \mathsf{type}}{\Delta; \Gamma \vdash M : B}\ \textsc{TyEq} \qquad \frac{\Delta \vdash \Gamma\ \mathsf{ctxt}}{\Delta; \Gamma \vdash * : 1}\ \textsc{Unit}$$

$$\frac{\Delta; \Gamma \vdash M : A \qquad \Delta; \Gamma \vdash N : B}{\Delta; \Gamma \vdash (M, N) : A \times B}\ \textsc{Pair} \qquad \frac{\Delta; \Gamma \vdash M : A \times B}{\Delta; \Gamma \vdash \pi_1 M : A}\ \textsc{Proj1} \qquad \frac{\Delta; \Gamma \vdash M : A \times B}{\Delta; \Gamma \vdash \pi_2 M : B}\ \textsc{Proj2} \qquad \frac{\Delta; \Gamma \vdash M : A}{\Delta; \Gamma \vdash \mathrm{inl}\ M : A + B}\ \textsc{Inl}$$

$$\frac{\Delta; \Gamma \vdash M : B}{\Delta; \Gamma \vdash \mathrm{inr}\ M : A + B}\ \textsc{Inr} \qquad \frac{\begin{array}{c}\Delta; \Gamma \vdash M : A + B \\ \Delta; \Gamma, x : A \vdash N_1 : C \\ \Delta; \Gamma, y : B \vdash N_2 : C\end{array}}{\Delta; \Gamma \vdash \mathrm{case}\ M\ \mathrm{of\ inl}\ x.N_1; \mathrm{inr}\ y.N_2 : C}\ \textsc{Case} \qquad \frac{\Delta; \Gamma, x : A \vdash M : B}{\Delta; \Gamma \vdash \lambda x.M : A \to B}\ \textsc{Abs}$$

$$\frac{\Delta; \Gamma \vdash M : A \to B \qquad \Delta; \Gamma \vdash N : A}{\Delta; \Gamma \vdash MN : B}\ \textsc{App} \qquad \frac{\Delta, i{:}s; \pi_{i:s}^* \Gamma \vdash M : A}{\Delta; \Gamma \vdash \Lambda i.M : \forall i{:}s.A}\ \textsc{UnivAbs} \qquad \frac{\Delta; \Gamma \vdash M : \forall i{:}s.A \qquad \Delta \vdash e : s}{\Delta; \Gamma \vdash M[e] : (\mathrm{id}_\Delta, e)^* A}\ \textsc{UnivApp}$$

$$\frac{\Delta; \Gamma \vdash M : (\mathrm{id}_\Delta, e)^* A \qquad \Delta \vdash e : s \qquad \Delta, i{:}s \vdash A\ \mathsf{type}}{\Delta; \Gamma \vdash \langle [e], M \rangle : \exists i{:}s.A}\ \textsc{ExPack} \qquad \frac{\Delta; \Gamma \vdash M : \exists i{:}s.A \qquad \Delta, i{:}s; \pi_{i:s}^* \Gamma, x : A \vdash N : \pi_{i:s}^* B}{\Delta; \Gamma \vdash \mathrm{let} \langle [i], x \rangle = M\ \mathrm{in}\ N : B}\ \textsc{ExUnpack}$$

Figure 5. Well-typed terms

interprets open terms as functions that map environments to final values.

Interpretation of Types. The defining feature of the index erasure interpretation is that semantics of a well-indexed type $\mathtt{X}\langle e_1, \ldots, e_n \rangle$ is determined solely by the primitive type $\mathtt{X}$ and not by the index expressions $e_1, \ldots, e_n$. We thus assume each primitive type $\mathtt{X} \in \mathit{PrimType}$ is assigned a set $\lfloor \mathtt{X} \rfloor$ and extend this assignment to well-indexed types by induction on the type structure:

$$\begin{array}{ll} \lfloor \mathtt{unit} \rfloor \stackrel{def}{=} \{*\} & \lfloor A + B \rfloor \stackrel{def}{=} \lfloor A \rfloor + \lfloor B \rfloor \\ \lfloor A \times B \rfloor \stackrel{def}{=} \lfloor A \rfloor \times \lfloor B \rfloor & \lfloor \mathtt{X}\langle e_1, \ldots, e_n \rangle \rfloor \stackrel{def}{=} \lfloor \mathtt{X} \rfloor \\ \lfloor A \to B \rfloor \stackrel{def}{=} \lfloor A \rfloor \to \lfloor B \rfloor & \lfloor \forall i{:}s.A \rfloor \stackrel{def}{=} \lfloor A \rfloor \\ & \lfloor \exists i{:}s.A \rfloor \stackrel{def}{=} \lfloor A \rfloor \end{array}$$

(We will overload the notation $\lfloor \cdot \rfloor$ for all index-erasure interpretations, and reserve notation $[\![\cdot]\!]$ for the index-observing relational semantics defined later.)

The index-erasure interpretation completely ignores index expressions and quantifiers, and type equality is defined as an extension of index equality. Therefore, it is straightforward to prove that equal types have equal denotations when interpreted in the index-erasure semantics, and that substitution of index terms has no effect on the index-erasure interpretation of types:

Lemma 1. *1. If* $\Delta \vdash A \equiv B\ \mathsf{type}$ *then* $\lfloor A \rfloor = \lfloor B \rfloor$*; and*
2. If $\Delta' \vdash A\ \mathsf{type}$ *and* $\Delta \vdash \sigma \Rightarrow \Delta'$*, then* $\lfloor \sigma^* A \rfloor = \lfloor A \rfloor$.

Interpretation of Terms. We assign an index-erasure semantics to any well-indexed typing context $\Delta \vdash \Gamma\ \mathsf{ctxt}$ by induction: $\lfloor \epsilon \rfloor = \{*\}$ and $\lfloor \Gamma, x : A \rfloor = \lfloor \Gamma \rfloor \times \lfloor A \rfloor$. For a well-typed term $\Delta; \Gamma \vdash M : A$, we define the *erasure interpretation* as a function $\lfloor M \rfloor : \lfloor \Gamma \rfloor \to \lfloor A \rfloor$ that completely ignores the indexing information. In light of Lemma 1, we do this directly on the syntax of well-typed terms, rather than on typing derivations. The definition of $\lfloor M \rfloor$ is completely standard, except for the clauses for universal and existential types:

$$\begin{array}{ll} \lfloor \Lambda i.\, M \rfloor\, \eta \stackrel{def}{=} \lfloor M \rfloor\, \eta & \lfloor M[e] \rfloor\, \eta \stackrel{def}{=} \lfloor M \rfloor\, \eta \\ \lfloor \langle [e], M \rangle \rfloor\, \eta \stackrel{def}{=} \lfloor M \rfloor\, \eta & \\ \lfloor \mathrm{let} \langle [i], x \rangle = M\ \mathrm{in}\ N \rfloor\, \eta \stackrel{def}{=} \lfloor N \rfloor\, (\eta, \lfloor M \rfloor\, \eta) & \end{array}$$

For any particular theory we assume that there is an interpretation of the primitive operations $\eta_{ops} \in \lfloor \Gamma_{ops} \rfloor$.

Example (Geometry: interpretation)**.** The two-dimensional geometry instantiation of the general framework uses the assignment $\lfloor \mathtt{vec} \rfloor = \mathbb{R}^2$ and $\lfloor \mathtt{real} \rfloor = \mathbb{R}$. We assume that η_{ops} gives the usual interpretation to scalar and vector operations from Figure 2 and Figure 3. □

Contextual Equivalence. We use our index-erasure semantics to define when a pair of terms are contextually equivalent with respect to syntactically defined contexts, following Hofmann [11]. Given an index context $\Delta = i_1{:}s_1, ..., i_n{:}s_n$, we write $\forall \Delta.A$ for $\forall i_1{:}s_1 \ldots \forall i_n{:}s_n.A$, and similarly for $\Lambda\Delta.M$ and $\lambda\Gamma.M$.

Definition 1 (Contextual Equivalence)**.** *Two terms* $\Delta; \Gamma_{ops}, \Gamma \vdash M_1, M_2 : A$ *are* contextually equivalent*, written* $\Delta; \Gamma \vdash M_1 \approx M_2 : A$*, if for all contexts* $\cdot; \Gamma_{ops} \vdash C : (\forall \Delta. \Gamma \to A) \to \mathtt{bool}$*, it is the case that* $\lfloor C\ (\Lambda\Delta.\lambda\Gamma.M_1) \rfloor\, \eta_{ops} = \lfloor C\ (\Lambda\Delta.\lambda\Gamma.M_2) \rfloor\, \eta_{ops}$.

Type Isomorphism. We say that well-indexed types $\Delta \vdash A\ \mathsf{type}$ and $\Delta \vdash B\ \mathsf{type}$ are isomorphic, and write $\Delta \vdash A \cong B$, if there exist maps between them that are mutually inverse with respect to contextual equivalence, i.e., if there are terms $\Delta \vdash I : A \to B$ and $\Delta \vdash J : B \to A$ such that $\Delta; x : A \vdash J(I(x)) \approx x : A$ and $\Delta; y : B \vdash I(J(y)) \approx y : B$. That $\cong$ is a congruence with respect to the type formation rules of Figure 4 is straightforward. We can also derive isomorphisms that are independent of the indexing theory, such as $\Delta \vdash A \times B \cong B \times A$, and $\Delta \vdash \forall i{:}s.(A \to B) \cong A \to \forall i{:}s.B$ for i not free in A.

3.4 The Relational Interpretation of Types

The relational semantics of the well-indexed type $\Delta \vdash A\ \mathsf{type}$ is a binary relation on the index-erasure interpretation of A. We write $\mathrm{Rel}(X)$ for the set of binary relations $R \subseteq X \times X$ on the set X.

For the unit, tuple, sum and function types we define the relational interpretation as a standard logical relation. The relational interpretations of primitive types with index arguments and the universally quantified types require an interpretation of index contexts.

An *index environment* ρ assigns to each index variable $i : s$ in the context Δ a value drawn from an interpretation of the sort s that soundly models the equational theory associated with s. We call such an interpretation a *model*: it assigns to each index operation in the equational theory a corresponding operation in the interpretation, so that index expressions can be interpreted by recursion on their structure.

For example, the sort T_2 of translations can be modelled by any abelian group. An obvious candidate here is the additive group over

$\mathbb{R}^2$, which we will use to obtain invariance under translation; but we could use the additive group $\mathbb{Q}$, or even a finite group such as the two-element group $\mathbb{Z}_2$.

Models. A *model* assigns to each sort $s \in \mathit{Sort}$ a carrier set $[\![s]\!]$, and assigns to each operation $\mathtt{f} \in \mathit{IndexOp}$ with $\mathrm{opArity}(\mathtt{f}) = ([s_1, \ldots, s_n], s)$, a function $[\![\mathtt{f}]\!] : [\![s_1]\!] \times \cdots \times [\![s_n]\!] \to [\![s]\!]$.

An index context $\Delta = (i_1{:}s_1, \ldots, i_n{:}s_n)$ is interpreted as cartesian product, i.e., $[\![\Delta]\!] \stackrel{def}{=} [\![s_1]\!] \times \cdots \times [\![s_n]\!]$. For each well-sorted index expression $\Delta \vdash e : s$, we assign a function $[\![e]\!] : [\![\Delta]\!] \to [\![s]\!]$ by recursion on the structure of e:

$$[\![i]\!]\rho \stackrel{def}{=} \rho(i) \qquad [\![\mathtt{f}(e_1, \ldots, e_n)]\!]\rho \stackrel{def}{=} [\![\mathtt{f}]\!]([\![e_1]\!]\rho, \ldots, [\![e_n]\!]\rho)$$

Finally, a model must be sound, that is, for each axiom $\Delta \vdash e \stackrel{ax}{\equiv} e' : s \in \mathit{IndexAx}$, we have $[\![e]\!] = [\![e']\!]$.

Example (Geometry: affine model)**.** We define the model of the indexing theory for the two-dimensional geometry example as follows. Each of the sorts is interpreted just as its semantic counterpart:

$$[\![\mathsf{T}_2]\!] = \mathrm{T}_2 \quad [\![\mathsf{GL}_2]\!] = \mathrm{GL}_2 \quad [\![\mathsf{O}_2]\!] = \mathrm{O}_2 \quad [\![\mathsf{GL}_1]\!] = \mathrm{GL}_1$$

Each of the index operations (e.g., the group structure and determinant) is interpreted by the intended semantic operation, and clearly satisfies the axioms in Example 3.1.2. □

Given $\rho \in [\![\Delta]\!]$ and a substitution $\Delta \vdash \sigma \Rightarrow \Delta'$ with $\sigma = (e_1, \ldots, e_n)$, we can derive the composed index environment $\rho \circ \sigma \in [\![\Delta']\!]$ as $\rho \circ \sigma \stackrel{def}{=} ([\![e_1]\!]\rho, \ldots, [\![e_n]\!]\rho)$.

Relational interpretation of primitive types. Having fixed a model we next choose a relational interpretation of primitive types: for each each primitive type $\mathtt{X}$, its relational interpretation is parameterised by elements from the model: $[\![\mathtt{X}]\!] : [\![\mathrm{tyArity}(\mathtt{X})]\!] \to \mathrm{Rel}(\lfloor \mathtt{X} \rfloor)$.

Example (Geometry: change of basis interpretation)**.** Given the model of geometric groups described above, for the relational interpretation of $\mathtt{vec}\langle B, t\rangle$ and $\mathtt{real}\langle s\rangle$, we use $[\![\mathtt{vec}]\!](B, \vec{t}) = \{(\vec{v}, B\vec{v} + \vec{t}) \mid \vec{v} \in \mathbb{R}^2\}$ and $[\![\mathtt{real}]\!](k) = \{(x, kx) \mid x \in \mathbb{R}\}$. □

Relational interpretation of types. We assign a relational interpretation to all well-indexed types $\Delta \vdash A\ \mathsf{type}$ by induction on their derivations, parameterised by index environments $\rho \in [\![\Delta]\!]$:

$$\begin{aligned}
[\![\mathtt{unit}]\!]\rho &\stackrel{def}{=} \{(*, *)\} \\
[\![\mathtt{X}\langle e_1, \ldots, e_n\rangle]\!]\rho &\stackrel{def}{=} [\![\mathtt{X}]\!]([\![e_1]\!]\rho, \ldots, [\![e_n]\!]\rho) \\
[\![A \to B]\!]\rho &\stackrel{def}{=} [\![A]\!]\rho \mathbin{\widehat{\to}} [\![B]\!]\rho \\
[\![A \times B]\!]\rho &\stackrel{def}{=} [\![A]\!]\rho \mathbin{\widehat{\times}} [\![B]\!]\rho \\
[\![A + B]\!]\rho &\stackrel{def}{=} [\![A]\!]\rho \mathbin{\widehat{+}} [\![B]\!]\rho \\
[\![\forall i{:}s.A]\!]\rho &\stackrel{def}{=} \bigcap\{[\![A]\!](\rho, m) \mid m \in [\![s]\!]\} \\
[\![\exists i{:}s.A]\!]\rho &\stackrel{def}{=} \bigcup\{[\![A]\!](\rho, m) \mid m \in [\![s]\!]\}
\end{aligned}$$

In this definition, the relational interpretation of an application of a primitive type $[\![\mathtt{X}\langle e_1, \ldots, e_n\rangle]\!]\rho$ is built from the relational interpretation of the primitive type, $[\![\mathtt{X}]\!]$, and the interpretation of the index terms $e_1, ..., e_n$ in the index environment ρ. Universal and existential quantification are interpreted by the set-theoretic intersection and union, respectively, over all extensions of the index environment.

We have also used the following standard constructions on binary relations: if $R \in \mathrm{Rel}(X)$ and $S \in \mathrm{Rel}(Y)$, then $R \mathbin{\widehat{\to}} S \in \mathrm{Rel}(X \to Y)$ is $\{(f_1, f_2) \mid \forall (a_1, a_2) \in R.\ (f_1 a_1, f_2 a_2) \in S\}$, and $R \mathbin{\widehat{\times}} S \in \mathrm{Rel}(X \times Y)$ is $\{((a_1, b_1), (a_2, b_2)) \mid (a_1, a_2) \in R \wedge (b_1, b_2) \in S\}$, and $R \mathbin{\widehat{+}} S \in \mathrm{Rel}(X + Y)$ is $\{(\mathrm{inl}\, x, \mathrm{inl}\, x') \mid (x, x') \in R\} \cup \{(\mathrm{inr}\, y, \mathrm{inr}\, y') \mid (y, y') \in S\}$.

The following lemma states that the relational interpretation of types that we have defined in this section behaves well: the first part of the lemma states that two types that are judgmentally equal are given equal relational interpretations, and the second part states that substitution of index expressions in types can be interpreted by the composition of index environments with substitutions.

Lemma 2. *1. If $\Delta \vdash A \equiv B\ \mathsf{type}$, then $[\![A]\!] = [\![B]\!]$;*
2. If $\Delta' \vdash A\ \mathsf{type}$ then for all $\Delta \vdash \sigma \Rightarrow \Delta'$ and $\rho \in [\![\Delta]\!]$, $[\![\sigma^* A]\!]\rho = [\![A]\!](\rho \circ \sigma)$.

Note that the equations in both parts of Lemma 2 are well-typed by virtue of the corresponding parts of Lemma 1.

3.5 The Abstraction Theorem and Semantic Equivalence

Our main result (Theorem 1) is that the index-erasure semantics of every well-typed term is related to itself in the relational interpretation of its type: this is the Abstraction Theorem for every instantiation of our general framework.

The Abstraction Theorem. We now state the Abstraction Theorem for well-typed terms. To state and prove this theorem for open terms, we extend the relational interpretation of types to typing contexts. The relational interpretation of contexts is defined by:

$$[\![\epsilon]\!]\rho \stackrel{def}{=} \{(*, *)\} \qquad [\![\Gamma, x : A]\!]\rho \stackrel{def}{=} [\![\Gamma]\!]\rho \mathbin{\widehat{\times}} [\![A]\!]\rho$$

The relational interpretation of contexts inherits from the relational interpretation of types the property of interpreting the application of substitutions as composition:

Lemma 3. *If $\Delta' \vdash \Gamma\ \mathsf{ctxt}$ and $\Delta \vdash \sigma \Rightarrow \Delta'$, then for all $\rho \in [\![\Delta]\!]$, we have $[\![\sigma^* \Gamma]\!]\rho = [\![\Gamma]\!](\rho \circ \sigma)$.*

Given a particular choice of model and relational interpretation of primitive types, we can then prove the following.

Theorem 1 (Abstraction)**.** *If $\Delta; \Gamma \vdash M : A$, then for all $\rho \in [\![\Delta]\!]$ and $\eta_1, \eta_2 \in \lfloor \Gamma \rfloor$ such that $(\eta_1, \eta_2) \in [\![\Gamma]\!]\rho$, we have $(\lfloor M \rfloor\, \eta_1, \lfloor M \rfloor\, \eta_2) \in [\![A]\!]\rho$.*

Proof. By induction on the typing derivation, making use of Lemma 2 (part 1) for rule TyEq, Lemma 2 (part 2) for rules UnivApp and ExPack, and Lemma 3 for rules UnivAbs and ExUnpack. The details can be found in the Coq development. □

Semantic equivalence of terms is defined in terms of the relational interpretation. As a consequence of Theorem 1, semantic equivalence is a sound approximation of contextual equivalence.

Let Γ_{ops} be the context of primitive operations and η_{ops} its interpretation Fix a model and relational interpretation of primitive types so that $(\eta_{ops}, \eta_{ops}) \in [\![\Gamma_{ops}]\!]*$.

Definition 2 (Semantic equivalence)**.** *Two terms $\Delta; \Gamma_{ops}, \Gamma \vdash M_1, M_2 : A$ are* semantically equal*, written $\Delta; \Gamma \models M_1 \sim M_2 : A$, if for all $\rho \in [\![\Delta]\!]$, and all $(\eta_1, \eta_2) \in [\![\Gamma]\!]\rho$, we have $(\lfloor M_1 \rfloor\, (\eta_{ops}, \eta_1), \lfloor M_2 \rfloor\, (\eta_{ops}, \eta_2)) \in [\![A]\!]\rho$.*

Theorem 2 (Soundness)**.** *If $\Delta; \Gamma \models M_1 \sim M_2 : A$ then $\Delta; \Gamma \vdash M_1 \approx M_2 : A$.*

4. Geometric Consequences of Abstraction

We now instantiate our general framework with the indexing theory of Section 2, and present more general and formally-justified free theorems, type isomorphisms, and non-definability results. For the free theorems and isomorphisms, we use the model and relational

interpretation of primitive types described in Section 3.4, namely that of affine transformations for vectors and scaling for scalars.

4.1 Free Theorems

Consider the type of the triangle area function from Example 4:

$$\begin{aligned}&\text{area} : \forall B{:}\mathsf{GL}_2, t{:}\mathsf{T}_2.\\ &\qquad \mathtt{vec}\langle B, t\rangle \to \mathtt{vec}\langle B, t\rangle \to \mathtt{vec}\langle B, t\rangle \to \mathtt{real}\langle|\det B|\rangle\end{aligned}$$

By Theorem 1, we can derive the following free theorem. For all $B \in \mathrm{GL}_2$, $\vec{t} \in \mathrm{T}_2$, and $\vec{x}, \vec{y}, \vec{z} \in \mathbb{R}^2$, we have

$$|\det B|(\lfloor \text{area} \rfloor\ \vec{x}\ \vec{y}\ \vec{z}) = \lfloor \text{area} \rfloor\ (B\vec{x} + \vec{t})\ (B\vec{y} + \vec{t})\ (B\vec{z} + \vec{t})$$

Thus, directly from the type of the area function, we can see that its index-erasure semantics is (a) invariant under translations, and (b) if the inputs are subjected to a linear transformation B, the output varies with the absolute value of the determinant of B.

4.2 Type Isomorphisms

Types indexed by abelian groups induce a particularly rich theory of type isomorphisms; previous work on units of measure [13] relates these to Buckingham's theorem from dimensional analysis. Here we consider the additive abelian groups of translations and the multiplicative abelian group of scalings.

Translations Consider first the group T_2 of translations from Section 2.2 and Section 2.3.

Example 7 (Geometry: wlog)**.** We prove that

$$\Delta, B{:}\mathsf{GL}_2 \vdash (\forall t{:}\mathsf{T}_2.\mathtt{vec}\langle B, t\rangle \to \mathtt{vec}\langle B, t\rangle) \cong \mathtt{vec}\langle B, 0\rangle.$$

Let

$$X \stackrel{def}{=} \forall t{:}\mathsf{T}_2.\mathtt{vec}\langle B, t\rangle \to \mathtt{vec}\langle B, t\rangle \quad \text{and} \quad Y \stackrel{def}{=} \mathtt{vec}\langle B, 0\rangle$$

$$I \stackrel{def}{=} \lambda f{:}X.f\ [0]\ (0\ [B])$$

$$J \stackrel{def}{=} \lambda y{:}Y.\Lambda t{:}\mathsf{T}_2.\lambda w{:}\mathtt{vec}\langle B, t\rangle.v + [B, 0, t]w$$

Unfolding definitions gives that $\lfloor I(J(v))\rfloor\ (\eta_{ops}, \vec{v}) = \vec{v} + \vec{0}$ for any $\vec{v} \in \mathbb{R}^2$. Because $\vec{0}$ is the identity for vector addition it follows that $\Delta, B{:}\mathsf{GL}_2; v{:}Y \vdash I(J(v)) \approx v : Y$. To show $\Delta, B{:}\mathsf{GL}_2; f{:}X \vdash J(I(f)) \approx f : X$ we appeal to Theorem 2 and reason using the relational semantics. It suffices to show $\Delta, B{:}\mathsf{GL}_2; f{:}X \models J(I(f)) \sim f : X$; that is, given $\rho \in [\![\Delta, B{:}\mathsf{GL}_2]\!]$, for any f and f' with $(f, f') \in [\![X]\!]\rho$, it is the case that $(\lfloor J(I(f))\rfloor\ (\eta_{ops}, f), f') \in [\![X]\!]\rho$. Expanding the premise, we have

$$\forall \vec{t} \in \mathbb{R}^2, \forall \vec{v} \in \mathbb{R}^2, f'(\rho(B)\vec{v} + \vec{t}) = \rho(B)(f(\vec{v})) + \vec{t} \qquad (3)$$

Expanding the conclusion, we have to show that for any $\vec{t_0} \in \mathbb{R}^2$ and $\vec{v_0} \in \mathbb{R}^2$, it is the case that $f'(\rho(B)\vec{v_0} + \vec{t_0}) = \rho(B)(\vec{v_0} + f(0)) + \vec{t_0}$. By instantiating Equation 3 with $\vec{v} = \vec{0}$ and $\vec{t} = \rho(B)\vec{v_0} + \vec{t_0}$ and applying a little algebra we obtain this result. □

We can generalise this isomorphism substantially, proving that for any quantifier-free type A, the type $\forall t{:}\mathsf{T}_2.\mathtt{vec}\langle B, t\rangle \to A$ is isomorphic to $A[0/t]$, where $A[0/t]$ denotes substitution of the identity translation 0 for index variable t in type A. This isomorphism formalises the equivalence between *coordinate-free* geometry, expressed by a polymorphic type whose first argument can be thought of as the *origin* with respect to which A is specified, and a *coordinate-based* geometry, expressed by the type $A[0/t]$, in which the origin is fixed at $(0, 0)$. It is sometimes said that an affine space is a vector space that has forgotten its origin; we have captured this in a type isomorphism.

Lemma 4. *Suppose that* $\Delta, B{:}\mathsf{GL}_2, t{:}\mathsf{T}_2 \vdash A$ type*,* A *contains no quantifiers and every occurrence of* vec *is of the form* $\mathtt{vec}\langle B, ?\rangle$*.*

Then

$$\Delta, B{:}\mathsf{GL}_2 \vdash (\forall t{:}\mathsf{T}_2.\mathtt{vec}\langle B, t\rangle \to A) \cong A[0/t]$$

Proof. Let

$$X \stackrel{def}{=} \forall t{:}\mathsf{T}_2.\mathtt{vec}\langle B, t\rangle \to A \quad \text{and} \quad Y \stackrel{def}{=} A[0/t]$$

$$I \stackrel{def}{=} \lambda f{:}X.f\ [0]\ (0\ [B])$$

$$J \stackrel{def}{=} \lambda y{:}Y.\Lambda t.\lambda v{:}\mathtt{vec}\langle B, t\rangle.\uparrow_A^{t:v}(y)$$

where terms

$$\Delta, B{:}\mathsf{GL}_2, t{:}\mathsf{T}_2; v{:}\mathtt{vec}\langle B, t\rangle \vdash \uparrow_A^{t:v} : A[0/t] \to A$$

$$\Delta, B{:}\mathsf{GL}_2, t{:}\mathsf{T}_2; v{:}\mathtt{vec}\langle B, t\rangle \vdash \downarrow_A^{t:v} : A \to A[0/t]$$

$$\Delta, B{:}\mathsf{GL}_2, t{:}\mathsf{T}_2; v{:}\mathtt{vec}\langle B, t\rangle \vdash \uparrow_e^{t:v} : \mathtt{vec}\langle B, e[0/t]\rangle \to \mathtt{vec}\langle B, e\rangle$$

$$\Delta, B{:}\mathsf{GL}_2, t{:}\mathsf{T}_2; v{:}\mathtt{vec}\langle B, t\rangle \vdash \downarrow_e^{t:v} : \mathtt{vec}\langle B, e\rangle \to \mathtt{vec}\langle B, e[0/t]\rangle$$

are defined by induction over the structure of A and e. For brevity we omit types on binders; also note that the omitted cases for $\downarrow_A^{t:v}$ have definition symmetric to $\uparrow_A^{t:v}$:

$$\begin{aligned}\uparrow_{\mathtt{unit}}^{t:v} &= \lambda x.x \qquad \uparrow_{\mathtt{real}\langle e\rangle}^{t:v} = \lambda x.x\\ \uparrow_{A_1\times A_2}^{t:v} &= \lambda p.(\uparrow_{A_1}^{t:v}(\pi_1 p), \uparrow_{A_2}^{t:v}(\pi_2 p))\\ \uparrow_{A_1+A_2}^{t:v} &= \lambda x.\text{case } x \text{ of inl } y.\text{inl}(\uparrow_{A_1}^{t:v} y); \text{inr } y.\text{inr}(\uparrow_{A_2}^{t:v} y)\\ \uparrow_{A_1\to A_2}^{t:v} &= \lambda f.\lambda x.\uparrow_{A_2}^{t:v}(f(\downarrow_{A_1}^{t:v} x))\end{aligned}$$

$$\begin{aligned}\uparrow_{\mathtt{vec}\langle B,e\rangle}^{t:v} &= \lambda x.\downarrow_e^{t:v}(x) & \downarrow_{\mathtt{vec}\langle B,e\rangle}^{t:v} &= \lambda x.\uparrow_e^{t:v}(x)\\ \uparrow_0^{t:v} &= \lambda w.w & \downarrow_0^{t:v} &= \lambda w.w\\ \uparrow_{e_1+e_2}^{t:v} &= \lambda w.\uparrow_{e_1}^{t:v}(\uparrow_{e_2}^{t:v}(w)) & \downarrow_{e_1+e_2}^{t:v} &= \lambda w.\downarrow_{e_1}^{t:v}(\downarrow_{e_2}^{t:v}(w))\\ \uparrow_{-e}^{t:v} &= \lambda w.\downarrow_e^{t:v}(w) & \downarrow_{-e}^{t:v} &= \lambda w.\uparrow_e^{t:v}(w)\\ \uparrow_t^{t:v} &= \lambda w.w + v & \downarrow_t^{t:v} &= \lambda w.w + (\text{negate } v)\\ \uparrow_{t'}^{t:v} &= \lambda w.w \quad (t' \neq t) & \downarrow_{t'}^{t:v} &= \lambda w.w \quad (t' \neq t)\end{aligned}$$

Unfolding definitions, we deduce that

$$\lfloor I(J(y))\rfloor\ (\eta_{ops}, y) = \lfloor \uparrow_A^{t:v} \rfloor\ (\eta_{ops}, \vec{0})(y)$$

for any $y \in \lfloor A[0/t]\rfloor$. It's easy to prove by induction on A that $\lfloor \uparrow_A^{t:v} \rfloor\ (\eta_{ops}, \vec{0})(y) = y$, and so $\Delta, B{:}\mathsf{GL}_2; y{:}Y \vdash I(J(y)) \approx y : Y$ follows.

For the opposite direction of the isomorphism, we appeal to Theorem 2 and so it suffices to show $\Delta, B{:}\mathsf{GL}_2; f{:}X \models J(I(f)) \sim f : X$; that is, given $\rho \in [\![\Delta, B{:}\mathsf{GL}_2]\!]$, for any f and f' with $(f, f') \in [\![X]\!]\rho$, it is the case that $(\lfloor J(I(f))\rfloor\ (\eta_{ops}, f), f') \in [\![X]\!]\rho$. Expanding the premise, we have

$$\forall \vec{t} \in \mathbb{R}^2, \forall \vec{v} \in \mathbb{R}^2, (f(\vec{v}), f'(\rho(B)\vec{v} + \vec{t})) \in [\![A]\!](\rho, \vec{t}) \qquad (4)$$

Expanding the conclusion, we have to show for any $\vec{t_0} \in \mathbb{R}^2$ and $\vec{v_0} \in \mathbb{R}^2$, it is the case that $(\lfloor J(I(f))\rfloor\ (\eta_{ops}, f)(\vec{v_0}), f'(\rho(B)\vec{v_0} + \vec{t_0})) \in [\![A]\!](\rho, \vec{t_0})$. Unfolding the definitions of J and I and expanding the erasure semantics, this amounts to showing that

$$\forall \vec{t_0}, \vec{v_0} \in \mathbb{R}^2, (\lfloor \uparrow_A^{t:v} \rfloor\ (\eta_{ops}, \vec{v_0})(f\ \vec{0}), f'(\rho(B)\vec{v_0} + \vec{t_0})) \in [\![A]\!](\rho, \vec{t_0}).$$

In order to make progress we prove first by induction on A that for any $\vec{v}$ and $\vec{w}$, the meaning of $\uparrow_A^{t:v}$ can be characterized by

$$\begin{aligned}&(\lfloor \uparrow_A^{t:v} \rfloor\ (\eta_{ops}, \vec{v})(x), y) \in [\![A]\!](\rho, \vec{w})\\ &\qquad \Leftrightarrow (x, y) \in [\![A]\!](\rho, \rho(B)\vec{v} + \vec{w})\end{aligned}$$

Thus our goal becomes

$$\forall \vec{t_0}, \vec{v_0} \in \mathbb{R}^2, (f\ \vec{0}, f'(\rho(B)\vec{v_0} + \vec{t_0})) \in [\![A]\!](\rho, \rho(B)\vec{v_0} + \vec{t_0}).$$

This is obtained immediately by instantiating Equation 4 with $\vec{v} = \vec{0}$ and $\vec{t} = \rho(B)\vec{v_0} + \vec{t_0}$. □

Example 2 and Example 7 are special cases of this isomorphism. Another instance is the type of vector addition:

$$\begin{aligned}
&\forall B{:}\mathsf{GL}_2, t_1, t_2{:}\mathsf{T}_2.\mathtt{vec}\langle B, t_1\rangle \to \mathtt{vec}\langle B, t_2\rangle \to \mathtt{vec}\langle B, t_1{+}t_2\rangle \\
\cong\quad &\forall B{:}\mathsf{GL}_2, t_1{:}\mathsf{T}_2.\mathtt{vec}\langle B, t_1\rangle \to \mathtt{vec}\langle B, t_1\rangle \quad \text{(by Lemma 4)} \\
\cong\quad &\forall B{:}\mathsf{GL}_2.\mathtt{vec}\langle B, 0\rangle \quad \text{(by Lemma 4)} \\
\cong\quad &\mathtt{unit} \quad \text{(by Theorem 1)}
\end{aligned}$$

In other words, the vector addition operation is the only inhabitant of its type!

Scalings For the group GL_1 of scalings we can treat a real-valued argument to a function as a unit of measure with which to scale the result. Although the argument might be zero, and this cannot be used for scaling, we still can obtain the following slightly more complicated isomorphism:

$$\forall s{:}\mathsf{GL}_1.\mathtt{real}\langle s\rangle \to A \quad\cong\quad A[1/s] \times \forall s{:}\mathsf{GL}_1.A \qquad (5)$$

The type of areaCircle from the introduction is one instance:

$$\begin{aligned}
&\forall s{:}\mathsf{GL}_1.\mathtt{real}\langle s\rangle \to \mathtt{real}\langle s^2\rangle \\
\cong\quad &\mathtt{real}\langle 1\rangle \times (\forall s{:}\mathsf{GL}_1.\mathtt{real}\langle s^2\rangle) \quad \text{(by Equation 5)} \\
\cong\quad &\mathtt{real}\langle 1\rangle \times \mathtt{unit} \quad \text{(by Theorem 1)} \\
\cong\quad &\mathtt{real}\langle 1\rangle \quad \text{(trivially)}
\end{aligned}$$

4.3 Non-definability

To prove non-definability results, the model and relational interpretation used in the previous two sections are not sufficient. This is true even for simple scalars with invariance under scaling. Consider the type $\forall s{:}\mathsf{GL}_1.\mathtt{real}\langle s^2\rangle \to \mathtt{real}\langle s\rangle$. There are many functions $f : \mathbb{R} \to \mathbb{R}$ that satisfy its relational interpretation, i.e., for which $f(k^2 \cdot x) = k \cdot f(x)$ for any $k \neq 0$. Consider

$$f(x) = \begin{cases} \sqrt{x} & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

for instance. Yet this type contains only the constant zero function, a fact that we can prove using a surprisingly simple model!

Example 8. If $\Gamma_{ops} \vdash f : \forall s{:}\mathsf{GL}_1.\ \mathtt{real}\langle s^2\rangle \to \mathtt{real}\langle s\rangle$ then $\lfloor f \rfloor = \lambda x.0$. To show this, take the model

$$\begin{aligned}
&[\![\mathsf{GL}_1]\!] \stackrel{def}{=} \{0,1\} \qquad [\![1]\!] \stackrel{def}{=} 0 \qquad [\![^{-1}]\!] \stackrel{def}{=} \mathrm{id} \\
&[\![\cdot]\!] \stackrel{def}{=} \lambda(x,y).(x+y) \bmod 2 \qquad [\![|-|]\!] \stackrel{def}{=} \mathrm{id},
\end{aligned}$$

in other words, the additive abelian group $(\mathbb{Z}_2, +)$. Now set

$$[\![\mathtt{real}]\!](z) = \begin{cases} \Delta_{\mathbb{R}} & \text{if } z = 0 \\ \{(0,0)\} & \text{if } z = 1 \end{cases}$$

It's easy to check that $(\eta_{ops}, \eta_{ops}) \in [\![\Gamma_{ops}]\!]*$.

We think of z as tracking whether or not exponents on scale parameters are divisible by 2: all the primitive operations in Figure 2 produce results with even exponents if their inputs have even exponents. But the type under consideration here does not.

By Theorem 1 we have that $(f, f) \in [\![\forall s{:}\mathsf{GL}_1.\ \mathtt{real}\langle s^2\rangle \to \mathtt{real}\langle s\rangle]\!]*$. Expanding the definitions, this means that for any $z \in [\![\mathsf{GL}_1]\!]$ and any $(x, y) \in [\![\mathtt{real}]\!]((z + z) \bmod 2)$, it is the case that $(f(x), f(y)) \in [\![\mathtt{real}]\!](z)$. Set $z = 1$. Then we have that if $x = y$ then $f(x) = f(y) = 0$. □

The particular model and relational interpretation used here is devised only to rule out the 'square root type' (it does not rule out a 'cube root type', for instance); nevertheless, it is possible to devise craftier interpretations that serve up both invariance and more general non-definability results. For example, we can set

$$\begin{aligned}
&[\![\mathsf{GL}_1]\!] \stackrel{def}{=} \mathrm{GL}_1 \times \mathbb{Q} \qquad [\![\cdot]\!] \stackrel{def}{=} (\cdot, +) \qquad [\![1]\!] \stackrel{def}{=} (1, 0) \\
&[\![^{-1}]\!] \stackrel{def}{=} (^{-1}, -) \qquad [\![|\cdot|]\!] \stackrel{def}{=} (|\cdot|, \mathrm{id})
\end{aligned}$$

and

$$[\![\mathtt{real}]\!](k, q) = \begin{cases} \{(x, kx) \mid x \in \mathbb{R}\} & \text{if } q \in \mathbb{Z} \\ \{(0,0)\} & \text{otherwise} \end{cases}$$

Example 9. Suppose we extend the operations Γ_{ops} of Figure 2 and corresponding η_{ops} with a square root operation typed by

$$\mathrm{sqrt} : \forall s{:}\mathsf{GL}_1.\mathtt{real}\langle s^2\rangle \to \mathtt{real}\langle s\rangle.$$

If $\Gamma_{ops} \vdash f : \forall s{:}\mathsf{GL}_1.\ \mathtt{real}\langle s^3\rangle \to \mathtt{real}\langle s\rangle$ then $\lfloor f \rfloor = \lambda x.0$. To show this, take the model

$$[\![\mathsf{GL}_1]\!] \stackrel{def}{=} \mathbb{Q} \quad [\![\cdot]\!] \stackrel{def}{=} + \quad [\![1]\!] \stackrel{def}{=} 0 \quad [\![^{-1}]\!] \stackrel{def}{=} - \quad [\![|\cdot|]\!] \stackrel{def}{=} \mathrm{id},$$

in other words, the additive abelian group $(\mathbb{Q}, +)$. Now set

$$[\![\mathtt{real}]\!](q) = \begin{cases} \Delta_{\mathbb{R}} & \text{if } \exists n, 2^n q \in \mathbb{Z} \\ \{(0,0)\} & \text{otherwise} \end{cases}$$

It's easy to check that $(\eta_{ops}, \eta_{ops}) \in [\![\Gamma_{ops}]\!]*$.

Here, q is tracking the exponents on scale parameters, with $[\![\mathtt{real}]\!](q)$ relating only zero to itself unless the denominator of q is a power of two (a so-called *dyadic* number). Applying the Abstraction Theorem and expanding, we have that for any $q \in \mathbb{Q}$ and any $(x, y) \in [\![\mathtt{real}]\!](3q)$, it is the case that $(f(x), f(y)) \in [\![\mathtt{real}]\!](q)$. Set $q = \frac{1}{3}$. Then we have that if $x = y$ then $f(x) = f(y) = 0$. □

5. Non-definability for the General Framework

The non-definability results in the previous section required the use of specially crafted models and relational interpretations. It is reasonable to ask whether or not there is a general method for constructing suitable models and relational interpretations to prove non-definability results. In this section, we show that this is possible for a large class of first-order types in any instance of our general framework.

In Example 8, we showed that the type $\forall s{:}\mathsf{GL}_1.\ \mathtt{real}\langle s^2\rangle \to \mathtt{real}\langle s\rangle$ only has trivial inhabitants. Intuitively, this is because the index of the result type (s) cannot be obtained from the index of the input type (s^2) using the abelian group operations and axioms. This observation can be used to give a sufficient condition for non-inhabitation for types of the form:

$$\forall i_1{:}s_1, \ldots, i_m{:}s_m.\ \mathtt{X}\langle e_1\rangle \to \ldots \to \mathtt{X}\langle e_n\rangle \to \mathtt{X}\langle e\rangle$$

Roughly speaking, if this type is inhabited, then it must be the case that the index expression e can be generated from the set of index expressions $\{e_1, ..., e_n\}$.

We assume that we are working with an instantiation of the general framework from Section 3 with a closed typing context Γ_{ops} describing the types of the primitive operations and a chosen index-erasure semantics. We will use a special relational interpretation built from the syntax of the indexing expressions to prove our general non-definability result. For simplicity, we assume that there is only one primitive type, $\mathtt{X}$, but the technique we describe here extends to the general case. We also assume that $\lfloor \mathtt{X} \rfloor$ is non-empty.

To state our general non-definability condition, we need to define the set of index expressions generated by some finite set of index expressions. Given a set S of index expressions that are all well-indexed in some index context Δ, we define $\mathrm{Gen}_0(S)$ to be the set of expressions that are built from the elements of S and the primitive index operations. To account for the equations between

indexing terms, we close under index expression equivalence to get the set $\mathrm{Gen}(S) = \{e \mid \exists s.\ \exists e' \in \mathrm{Gen}_0(S).\ \Delta \vdash e \equiv e' : s\}$. Now a type is *well-generated* if it is closed and of the form:

$$\forall i_1{:}s_1, \ldots, i_m{:}s_m.\ \mathtt{X}\langle e_1\rangle \to \ldots \to \mathtt{X}\langle e_n\rangle \to \mathtt{X}\langle e\rangle$$

and $e \in \mathrm{Gen}(\{e_1, ..., e_n\})$.

Theorem 3. *Assume that the members of Γ_{ops} are all of well-generated types. If there exists an M with typing:*

$$\Gamma_{ops} \vdash M : \forall i_1{:}s_1, ..., i_m{:}s_m.\ \mathtt{X}\langle e_1\rangle \to ... \to \mathtt{X}\langle e_n\rangle \to \mathtt{X}\langle e\rangle$$

then $e \in \mathrm{Gen}(\{e_1, ..., e_n\})$.

This theorem is usually applied in the contrapositive: if $e \notin \mathrm{Gen}(\{e_1, ..., e_n\})$ then no such M can exist. Note that if Γ_{ops} contains operations corresponding to each of the index-level operations (as, for example, in Figure 2), then this theorem yields a characterisation of definable terms, since the construction of e from $e_1, ..., e_n$ can be replicated at the term level.

Proof. Let $\Delta = i_1{:}s_1, ..., i_m{:}s_m$ be the index context constructed from the universally quantified type variables in the type of M. To interpret the index expressions, we use the free model over the variables in Δ constructed from the syntax. This model assigns to each sort s the set $\{e \mid \Delta \vdash e : s\}/\equiv$ of index expressions quotiented by index expression equality. Index operations are interpreted by the corresponding syntactic operation on equivalence classes: $[\![\mathtt{f}]\!]([e_1], ..., [e_n]) = [\mathtt{f}(e_1, ..., e_n)]$.

We take the relational interpretation of the primitive type $\mathtt{X}$ as:

$$[\![\mathtt{X}]\!](e) = \{(x, x) \mid x \in \lfloor \mathtt{X} \rfloor \wedge e \in \mathrm{Gen}(\{e_1, ..., e_n\})\}$$

It is straightforward to check that for any index-erasure interpretation of the primitive operations $\eta_{ops} \in \Gamma_{ops}$, we have $(\eta_{ops}, \eta_{ops}) \in [\![\Gamma_{ops}]\!]*$ because all members of Γ_{ops} have well-generated types. Hence, by Theorem 1, we know that for all terms (i.e., elements of the free model over Δ) $e'_1{:}s_1, ..., e'_m{:}s_m$:

$$\begin{array}{l}\forall (x_1, x'_1) \in [\![\mathtt{X}\langle e_1\rangle]\!](e'_1, ..., e'_m), ...,\\ \quad (x_n, x'_n) \in [\![\mathtt{X}\langle e_n\rangle]\!](e'_1, ..., e'_m).\\ \quad (\lfloor M \rfloor\ \eta_{ops}\ x_1\ ...\ x_n, \lfloor M \rfloor\ \eta_{ops}\ x'_1\ ...\ x'_n) \in [\![\mathtt{X}\langle e\rangle]\!](e'_1, ..., e'_m)\end{array}$$

By setting $e'_j = i_j$, and using an arbitrary element $x \in \lfloor \mathtt{X} \rfloor$ (which we have assumed non-empty), we have, for all k, $(x, x) \in [\![\mathtt{X}\langle e_k\rangle]\!](i_1, ..., i_m)$ since each e_k is a member of the set we are using to generate terms. Now $(\lfloor M \rfloor\ \eta_{ops}\ x ... x, \lfloor M \rfloor\ \eta_{ops}\ x ... x) \in [\![\mathtt{X}]\!](e)$ and so $e \in \mathrm{Gen}(\{e_1, ..., e_n\})$. □

Application to Example 3 Theorem 3 can be directly applied to show that the type $\forall t{:}\mathsf{T}_2.\ \mathtt{vec}\langle t + t\rangle \to \mathtt{vec}\langle t\rangle$ has no inhabitants. The free model over the single index variable t is (isomorphic to) the integers, and the sub-model generated by the index expression $t + t$ corresponds to the even integers. The result now follows simply because 1 (i.e., the interpretation of t) is not an even number.

Abelian Group Indexed Types Kennedy [13] has given a general characterisation of definability at first-order in the case of abelian group indexing in terms of integer solutions to a set of linear equations. Specialising Theorem 3 to the case of abelian group indexing yields Kennedy's characterisation.

Polymorphic Constants Theorem 3 does not apply in the case when we have polymorphic constants. This is the case with the polymorphic $0 : \forall s{:}\mathsf{GL}_1.\ \mathtt{real}\langle s\rangle$ in Figure 2. Theorem 3 does not apply because the index expression s is not generated by the empty set: 0's type is not well-generated. Nevertheless, it is easy to adapt the proof of Theorem 3 to handle a polymorphic constant like 0 by setting the relational interpretation of $\mathtt{X}$ to be:

$$[\![X]\!](e) = \{(x, x) \mid x \in \lfloor \mathtt{X} \rfloor \wedge (x = 0 \vee e \in \mathrm{Gen}(\{e_1, ..., e_n\}))\}$$

The conclusion of the theorem now states that either we have $e \in \mathrm{Gen}(\{e_1, ..., e_n\})$ *or* $\lfloor M \rfloor$ is the constant 0 function. This extended theorem can now be used to give an alternative proof for Example 8. As this example illustrates, there may be many different models that can be used to prove a non-definability result.

Adding Index Operations Theorem 3 also does not directly apply in the case of Example 9, again because assumption that the types of the primitive operations are all well-generated is not satisfied. In this case, the assumed square root operation has type $\forall s{:}\mathsf{GL}_1.\ \mathtt{real}\langle s^2\rangle \to \mathtt{real}\langle s\rangle$, and as we observed at the start of this section, s is not in the set generated by s^2. However, to enable the application of the theorem, we can assume an additional index operation $-^{1/2}$, acting like square root at the index level. Now the free model produced in the proof of the theorem is isomorphic to the dyadic numbers with addition and halving, and the generated sub-model consists of the dyadic rationals of the form $\frac{3k}{2^n}$. Again there is a diversity of models that can be used to prove a single non-definability result.

6. Logical Information Flow

We now apply our general framework to types that are indexed by logical propositions. By including a primitive type that represents logical truth, we can recover–through a construction due to Tse and Zdancewic [21]–strong information flow properties of programs. As a result of our general framework being parameterised by the choice of equational theory, we can alter the logic that we use for reasoning about type equality, and hence alter the information flow properties of the system.

We first recall the concept of information flow. A function $f : A \times B \to C$ is said to not allow information to flow from its second argument to the output if for all $b, b' \in B$ and all $a \in A$, $f(a, b) = f(a, b')$. If we think of the B argument as representing high-security information, then we have stated that f does not allow the high-security input to flow to the low security output. Information flow can be seen as a kind of invariance property of programs, and so our relational interpretation of types is well tailored to proving this kind of property.

As described by Sabelfeld and Sands [19], information flow can be captured semantically by partial equivalence relations (PERs). Abadi, Banerjee, Heintze and Riecke [1] built a *Core Calculus for Dependency*, using a type system based around a security level indexed monad $T_l A$, using PERs to prove the information flow properties. Tse and Zdancewic [21] translated Abadi *et al.*'s calculus into System F[1] translating the monadic type $T_l A$ to $\alpha_l \to A$ for some free type variable α_l, and using Reynolds' Abstraction Theorem to prove information flow properties. For example, if the type variable α_H represents high-level information, then the non-interference property of the function f could be expressed by the System F type $A \to (\alpha_H \to B) \to C$. If a program cannot generate a value of type α_H, then it cannot access the value of B, and hence is insensitive to the actual value. Relationships between security levels are captured by postulating functions $\alpha_{l_1} \to \alpha_{l_2}$ whenever l_1 is a lower security level than l_2.

Using algebraic indexing, we refine Tse and Zdancewic's translation by replacing each type variable α_l with a primitive type $\mathtt{T}\langle \phi_l\rangle$ of representations of the truth of a logical proposition ϕ_l that stands for the security level l. The relationships between security levels are now replaced by logical entailment, so we only have functions of type $\mathtt{T}\langle \phi_{l_1}\rangle \to \mathtt{T}\langle \phi_{l_2}\rangle$ when ϕ_{l_1} entails ϕ_{l_2}. We instantiate our general relational framework to interpret $\mathtt{T}\langle \phi\rangle$ as the identity rela-

[1] Tse and Zdancewic's translation did not satisfy all the properties that they claimed, as pointed out by Shikuma and Igarashi [20]. However, this problem is not relevant to our discussion here.

tion if ϕ is true and the empty relation if ϕ is false. We shall see that this recovers the information flow properties of Abadi *et al.* and Tse and Zdancewic.

Instantiation of the General Framework We assume a single indexing sort prop and assume the operations and equations of boolean algebra. Thus we have constants $\top, \bot$ and binary operators $\wedge, \vee$ with the axioms of a bounded lattice, and a unary complementation operator $\neg$. We will use ϕ, ψ to stand for index expressions of sort prop. We use an equational presentation of boolean algebras to fit with our general framework, but note that we can define an order on index expressions as $\phi \leq \psi$ when $\phi = \phi \wedge \psi$.

We have a single primitive type $\mathtt{T}$, with $\mathrm{tyArity}(\mathtt{T}) = [\mathsf{prop}]$ and index-erasure semantics $\lfloor \mathtt{X} \rfloor = \{*\}$. Thus values of type $\mathtt{T}\langle\phi\rangle$ have no run-time content; their only meaning is given by the relational semantics. For the model of the indexing theory, we take an arbitrary boolean algebra L. The relational interpretation of the truth representation type is $[\![\mathtt{T}]\!](x) = \{(*,*) \mid x = \top\}$, where $\top$ is the top element of the boolean algebra L. The primitive operations Γ_{Log} reflect logical consequence:

$$\mathrm{truth} : \mathtt{T}\langle\top\rangle \qquad \mathrm{and} : \forall p, q{:}\mathsf{prop}.\ \mathtt{T}\langle p\rangle \to \mathtt{T}\langle q\rangle \to \mathtt{T}\langle p \wedge q\rangle$$
$$\mathrm{up} : \forall p, q{:}\mathsf{prop}.\ \mathtt{T}\langle p \wedge q\rangle \to \mathtt{T}\langle p\rangle$$

The combination of the TyEq rule and the primitive up operation allow for logical entailment to be reflected in programs: if we have $\phi \leq \psi$ and $M : \mathtt{T}\langle\phi\rangle$ then $\mathrm{up}\ M : \mathtt{T}\langle\psi\rangle$. Each of the primitive operations has a trivial interpretation, due to the index-erasure interpretation of $\mathtt{T}\langle\phi\rangle$ as a one-element set, giving an environment $\eta_{Log} \in \lfloor\Gamma\rfloor$. Less trivially, we have this lemma:

Lemma 5. $(\eta_{Log}, \eta_{Log}) \in [\![\Gamma_{Log}]\!]*$.

Information Flow We think of logical expressions as "composite principals". That is, propositional variables representing atomic principals that are combined using the logical connectives. We interpret "truth" for principals as stating that a principal is true when satisfied with the current state of affairs. Thus a relationship $\phi \leq \psi$ indicates that satisfaction of the composite principal ϕ implies satisfaction of the composite principal ψ. In terms of security levels, the ordering is reversed: if a high security principal is satisfied, then all of their subordinates must also be satisfied.

We adapt Tse and Zdancewic's translation of Abadi *et al.*'s monadic type to our setting. We define a type abbreviation $T_\phi A = \mathtt{T}\langle\phi\rangle \to A$, where A is a type and ϕ is an expression of sort prop. For every ϕ, we can endow the types $T_\phi -$ with the structure of a monad. This is due to the fact that it is an instance of the "environment" (or "reader") monad [12]. We read the types $T_\phi A$ as data of type A "protected" by the principal ϕ.

As a consequence of the relational interpretation given above, it follows that if we have an index expression ϕ that is interpreted as some value other than $\top$ in an index environment ρ, then for all $x, x' \in \lfloor T_\phi \mathtt{bool}\rfloor$, we have $(x, x') \in [\![T_\phi \mathtt{bool}]\!]\rho$. Thus if a principal is dissatisfied (i.e., $\phi \neq \top$), then data protected by this principal is indistinguishable from any other data, and a program cannot get access to the exact value. From this observation, and Theorem 1, we obtain the following information flow result:

Theorem 4. *Let ϕ and ψ be index expressions of sort* prop *in some indexing context Δ, such that $\psi \not\leq \phi$. Then for all terms:*

$$\Delta; \Gamma_{Log}, \Gamma \vdash M : T_\phi \mathtt{bool} \to T_\psi \mathtt{bool}$$

and all terms N_1, N_2 of type $T_\phi \mathtt{bool}$, $M\ N_1 \overset{ctx}{\approx} M\ N_2$. Thus there is no information flow from M's input to its output.

Note that if $\psi \leq \phi$, then it is always possible to write the identity function with this type, using the up operation. The theorem also holds if we move to logics other than boolean logic. For example, if our equational theory models intuitionistic logic by taking the axioms of Heyting algebras, then the same non-flow property for programs of type $T_{p \vee \neg p}\mathtt{bool} \to T_\top \mathtt{bool}$ holds, due to the lack of excluded middle. If we take linear logic, then programs of type $T_{p \otimes p}\mathtt{bool} \to T_p \mathtt{bool}$ have no information flow from their input, due to non-provability of $p \vdash p \otimes p$.

7. Distance-Indexed Types

The type system for geometry we discussed in Section 2 and Section 4 made use of a relational interpretation of primitive types that relates pairs of elements by some transformation if applying the transformation to the first element of a pair yields the second. Thus, the free theorems that we derived directly take the form of "invariance" properties, where some equation holds between two terms. In this section, we examine another instantiation of our general framework that relates values when they are within a certain distance. The free theorems that we obtain inform us of the effect that programs have on the distances between values. For example, a program M of type:

$$\forall \epsilon_1, \epsilon_2{:}\mathsf{R}^{>0}.\ \mathtt{real}\langle\epsilon_1\rangle \to \mathtt{real}\langle\epsilon_2\rangle \to \mathtt{real}\langle\epsilon_1 + \epsilon_2\rangle$$

must satisfy the property that for all $\epsilon_1, \epsilon_2 > 0$ and $x, x', y, y' \in \mathbb{R}$:

$$\begin{array}{l}\text{if } |x - x'| < \epsilon_1 \text{ and } |y - y'| < \epsilon_2 \text{ then} \\ \quad |\lfloor M\rfloor\ \eta_{Met}\ x\ y - \lfloor M\rfloor\ \eta_{Met}\ x'\ y'| < \epsilon_1 + \epsilon_2\end{array}$$

Instantiation of the General Framework We assume a single indexing sort $\mathsf{R}^{>0}$ to represent positive, non-zero real numbers. For the index operations, we assume the operations $\min, \max, +$ and multiplication by constant reals. There is a single primitive type $\mathtt{real}$ with $\mathrm{tyArity}(\mathtt{real}) = [\mathsf{R}^{>0}]$. The primitive operations Γ_{Met} are as follows, where c stands for arbitrary real-valued constants:

$$\begin{array}{rl}\underline{c} & : \forall \epsilon{:}\mathsf{R}^{>0}.\ \mathtt{real}\langle\epsilon\rangle \\ (+) & : \forall \epsilon_1, \epsilon_2{:}\mathsf{R}^{>0}.\ \mathtt{real}\langle\epsilon_1\rangle \to \mathtt{real}\langle\epsilon_2\rangle \to \mathtt{real}\langle\epsilon_1 + \epsilon_2\rangle \\ (-) & : \forall \epsilon_1, \epsilon_2{:}\mathsf{R}^{>0}.\ \mathtt{real}\langle\epsilon_1\rangle \to \mathtt{real}\langle\epsilon_2\rangle \to \mathtt{real}\langle\epsilon_1 + \epsilon_2\rangle \\ \underline{c}* & : \forall \epsilon{:}\mathsf{R}^{>0}.\ \mathtt{real}\langle\epsilon\rangle \to \mathtt{real}\langle c\epsilon\rangle \\ \mathrm{up} & : \forall \epsilon_1, \epsilon_2{:}\mathsf{R}^{>0}.\ \mathtt{real}\langle\epsilon_1\rangle \to \mathtt{real}\langle\max(\epsilon_1, \epsilon_2)\rangle\end{array}$$

We assume that the index-erasure semantics of the $\mathtt{real}$ type is just the set $\mathbb{R}$, so all except the last operation have straightforward interpretations. The up operation is interpreted just as the identity function. The index-erasure interpretations of the primitive operations are collected together into an environment $\eta_{Met} \in \lfloor\Gamma_{Met}\rfloor$.

For the relational intepretation we construct a model of the indexing theory by interpreting $\mathsf{R}^{>0}$ with strictly positive real numbers. We set $[\![\mathtt{real}]\!](\epsilon) = \{(x, x') \mid |x - x'| < \epsilon\}$.

Lemma 6. $(\eta_{Met}, \eta_{Met}) \in [\![\Gamma_{Met}]\!]*$.

Uniform continuity Using existential types, the standard ϵ-δ definition of *uniform* continuity can be expressed as $\forall \epsilon{:}\mathsf{R}^{>0}.\ \exists \delta{:}\mathsf{R}^{>0}.\ \mathtt{real}\langle\delta\rangle \to \mathtt{real}\langle\epsilon\rangle$. For any program M of this type, Theorem 1 gives a free theorem that is exactly uniform continuity:

$$\forall \epsilon{>}0.\exists \delta{>}0.\forall x, x'.|x{-}x'|{<}\delta \Rightarrow |\lfloor M\rfloor\ \eta_{Met}\ x{-}\lfloor M\rfloor\ \eta_{Met}\ x'|{<}\epsilon$$

This definition differs from the ϵ-δ definition of (regular) continuity by the order of quantification: there, $\forall x$ comes before $\exists \delta$, so the distance δ may depend on x. We suspect that to express standard continuity as a type would require some form of type dependency. Chaudhuri, Gulwani and Lublinerman [8] have given a program logic based approach to verifying the continuity of programs.

Function Sensitivity A type system with a relational interpretation that tracks distances between values has been investigated by Reed and Pierce [17] in the setting of differential privacy. Their system uses a linear type discipline to ensure that all programs are c-sensitive (i.e., the distance between the outputs is no greater than c times the distance between the inputs, for some constant c). We

can express their central concept of c-sensitivity (for functions on the reals) as an algebraically indexed type: $\forall\epsilon{:}\mathsf{R}^{>0}.\ \mathtt{real}\langle\epsilon\rangle \to \mathtt{real}\langle\frac{1}{c}\epsilon\rangle$. Investigating the precise connection between their system and ours is left to future work.

8. Discussion

We presented a general framework for algebraically indexed types and instantiated it to yield novel type systems for geometry, logical information flow and distance-indexed types. Our framework further demonstrates the power of relational reasoning about typed programs. From Theorem 1, we derived interesting free theorems, type isomorphisms and non-definability results. We conclude with some observations and suggestions for further work.

Further Applications and Extensions We have covered several applications of algebraically indexed types in this paper, but there are undoubtedly many more. Geometry for dimensions greater than two is an obvious candidate, as are systems that are invariant under different geometric groups (e.g., the Poincaré group for relativity). Mathematical Physics is particularly rich in theories that have some notion of invariance, and it will be exciting to pin down the precise connections between these and type systems for which an Abstraction Theorem holds. Cardelli and Gardner describe a process calculus that builds in 3D affine geometry [6], proving that process behaviour is invariant under affine transformations. Distinguishing points from vectors provides the appropriate abstraction barrier, and the geometric group is determined by inspecting term syntax. It would be interesting to recast their language in terms of our indexed types to obtain purely type-based invariance theorems.

Geometric theorem proving is another application. Harrison [10] comments on the pervasiveness of invariance properties in this area. Programs in our framework automatically satisfy invariance properties, removing the need for *ad hoc* proofs of these facts.

Type and effect analyses use types indexed by effect annotations with algebraic structure (e.g., sets of read/write effect labels with an idempotent monoid structure). Benton *et al.* have used relational interpretations to prove effect-dependent equivalences [4]. An extension of our framework with type-indexed types should be able to express their effect-indexed monads and prove their equivalences.

Extending our framework with type dependency would also allow for further applications. For example, we could consider a type of lists of length n, indexed by elements of the permutation group S_n. Bernardy *et al.* have presented a general framework for relational reasoning and an Abstraction Theorem for dependent types [5]. However, they work with pure type systems, which define type equality via untyped rewriting, so it is not immediately obvious how to integrate arbitrary equational theories into their framework.

Semantic Equality In general, the semantic equality in Definition 2 is not an equivalence relation. If the interpretations of all primitive types are partial equivalence relations then semantic equality is indeed an equivalence relation. However, this excludes the geometry and distance-indexed examples. More generally, we can consider relational interpretations that are *difunctional*. (A relation is difunctional if whenever (x, y), (x', y') and (x, y') are in the relation then so is (x', y).) Difunctionality is weaker than being a PER, but still suffices to prove that semantic equality is an equivalence relation. Hofmann [11] has used difunctional relations in the setting of effect analyses. Difunctionality covers all our examples except distance-indexed types. Note that for both PERs and difunctional relations we need to close the relational interpretation of existential types under the appropriate property to ensure that all types are interpreted as PERs/difunctional relations. For distance-indexed types it is possible that a new notion of equivalence based on closeness is required.

Acknowledgements

The authors would like to thank Nick Benton, Kenneth MacKenzie, John Reppy and Martin Will for illuminating discussions on geometry and types. Atkey and Johann were supported by EPSRC grant EP/G068917/1.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A Core Calculus of Dependency. Proceedings, POPL, pp.147-160, 1999.

[2] R. Atkey. Syntax for Free: Representing Syntax with Binding Using Parametricity. Proceedings, TLCA, pp. 35-49, 2009.

[3] N. Benton, C.-K. Hur, A. J. Kennedy, C. McBride. Strongly Typed Term Representations in Coq. *Journal of Automated Reasoning* 49(2), pp. 141-159, 2012.

[4] N. Benton, A. Kennedy, M. Hofmann, L. Beringer. Reading, Writing and Relations. Proceedings, APLAS, pp. 114-130, 2006.

[5] J.-P. Bernardy, P. Jansson, R. Paterson. Proofs for Free: Parametricity for Dependent Types. *Journal of Functional Programming* 22(2), pp. 107-152, 2012.

[6] L. Cardelli, P. Gardner. Processes in Space. *Programs, Proofs, Processes*: Proceedings, CiE, pp. 78-87, 2010.

[7] Computational Geometry Algorithms Library (CGAL): User and Reference Manual. Available at `http://www.cgal.org`.

[8] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity Analysis of Programs. Proceedings, POPL, pp. 57-70, 2010.

[9] J. Gallier. *Geometric Methods and Applications For Computer Science and Engineering*. Springer, 2011.

[10] J. Harrison. Without Loss of Generality. Proceedings, TPHOLs, pp. 43-59, 2009.

[11] M. Hofmann. Correctness of Effect-based Program Transformations. Formal Logical Methods for System Security and Correctness, pp. 149-173, 2008.

[12] M. P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. Proceedings, AFP, pp. 97-136, 1995.

[13] A. J. Kennedy. Relational Parametricity and Units of Measure. Proceedings, POPL, pp. 442-455, 1997.

[14] A. J. Kennedy. Types for Units-of-Measure: Theory and Practice. Central European Functional Programming school (CEFP), pp. 268–305, LNCS vol. 6299, 2010.

[15] S. Mann, N. Litke, T. DeRose. A Coordinate Free Geometry ADT. Technical Report CS-97-15, University of Waterloo, 1997.

[16] A. M. Pitts. Parametric Polymorphism and Operational Equivalence. *Mathematical Structures in Computer Science* 10(3), pp. 321-359, 2000.

[17] J. Reed and B. C. Pierce. Distance Makes the Types Grow Stronger. Proceedings, ICFP, pp. 157-169, 2010.

[18] J. C. Reynolds. Types, Abstraction and Parametric Polymorphism. *Information Processing* 83, pp. 513-523, 1983.

[19] A. Sabelfeld and D. Sands. A PER Model of Secure Information Flow in Sequential Programs. *Higher-Order and Symbolic Computation* 14 (1), pp. 59-91, 2001.

[20] N. Shikuma and A.Igarahsi. Proving Noninterference by a Fully Complete Translation to the Simply Typed lambda-calculus. *Logical Methods in Computer Science* 4(3), 2008.

[21] S. Tse and S. Zdancewic. Translating Dependency into Parametricity. Proceedings, ICFP, pp. 115-125, 2004.

[22] P. Wadler. Theorems for Free!. Proceedings, FPCA, pp. 347-359, 1989.