An Extensible Theory of Indexed Types

Daniel R. Licata  
*Carnegie Mellon University*

Robert Harper  
*Carnegie Mellon University*

Follow this and additional works at: [http://repository.cmu.edu/compsci](http://repository.cmu.edu/compsci)
Abstract
Indexed families of types are a way of associating run-time data with compile-time abstractions that can be used to reason about them. We propose an extensible theory of indexed types, in which programmers can define the index data appropriate to their programs and use them to track properties of run-time code. The essential ingredients in our proposal are (1) a logical framework, which is used to define index data, constraints, and proofs, and (2) computation with indices, both at the static and dynamic levels of the programming language. Computation with indices supports a variety of mechanisms necessary for programming with extensible indexed types, including the definition and implementation of indexed types, meta-theoretic reasoning about indices, proof-producing run-time checks, computed index expressions, and run-time actions of index constraints.

Programmer-defined index propositions and their proofs can be represented naturally in a logical framework such as LF, where variable binding is used to model the consequence relation of the logic. Consequently, the central technical challenge in our design is that computing with indices requires computing with the higher-order terms of a logical framework. The technical contributions of this paper are a novel method for computing with the higher-order data of a simple logical framework and the integration of such computation into the static and dynamic levels of a programming language. Further, we present examples showing how this computation supports indexed programming.

1. Introduction
One way to enrich the expressiveness of a type system is to provide the programmer with a rich language of data that can be used, at compile-time, to state and verify properties of run-time code. Compile-time data can be associated with run-time values using an indexed family of types whose indices vary with the classified values. For example, a programmer might define a family of types list[n:nat], indexed by a natural number representing the length of the list, and varying with the list constructors:

\[
\begin{align*}
nil & : \text{list}[0] \\
\text{cons} & : \forall n: \text{nat}. \text{elt} \rightarrow \text{list}[n] \rightarrow \text{list}[n+1]
\end{align*}
\]

Examples of indexing include dependent types, where the indices are drawn from the run-time programming language (Augustsson, 1998; Coquand and Huet, 1988; Flanagan, 2006; McBride and McKinna, 2004), and generalized algebraic datatypes, where the indices are other types (Cheney and Hinze, 2003; Peyton Jones et al., 2006; Sheard, 2004; Xi et al., 2003), as well as types indexed by static constraint domains (Chen and Xi, 2005; Dunfield and Pfenning, 2004; Fogarty et al., 2007; Licata and Harper, 2005; Sarkar, 2005; Xi and Pfenning, 1998), by propositions (Nanevski et al., 2006), and by proofs.

Indices serve as modelling types in the sense of Leino and Müller (2004), in that they define an abstraction of program values which may be used for reasoning. With dependent types, the available modelling types are the same as the values they model, and data is often used as its own model. Using more general indexing, one can model a value with abstractions other than the value itself, and the model need not be drawn from the run-time language. For example, static array bounds checking, as in Xi and Pfenning (1998), is possible using a family of types array[i:nat], which does not track the contents of the array but only its length i. Another example is Sarkar’s use of the types term[e:tm] and typ[t:tp], which index run-time datatypes representing the types and terms of a programming language by their LF (Harper et al., 1993) representations. The key application of indexing is that index information may be used to state properties of an abstract data type in its interface. For example, Sarkar writes a certified type checker, employing the LF representation of the language’s typing judgement as an index constraint:

\[
\begin{align*}
\text{check} & : \forall e. \forall t. \text{term}[e] \rightarrow \text{typ}[t] \\
& \rightarrow (\exists d: \text{of e t}. \text{unit}) + \text{unit}
\end{align*}
\]

When this function returns true, it also returns a certificate, represented as an LF derivation of the judgement of f e t, that the program is well-typed. Indexed types enable richer interfaces at module boundaries, serve as machine-checked documentation, and obviate some run-time checks. Proving that a program possesses a more precise type can be harder, but in return the type tells more about the program’s behavior. Pragmatically, the programmer can use indexing inasmuch as it seems worthwhile to capture such strong invariants.

An Extensible Theory of Indexing. We wish to provide an extensible theory of indexing, in which programmers can define their own index domains and use them to give stronger interfaces. An extensible framework must at least provide the ability to define index domains, index expressions inhabiting them, constraints on indices, proofs of constraints, indexed families of types computed by analysis of indices, and inhabitants of such families computed by such analysis. This can be achieved by equipping a functional language with a logical framework that is sufficiently powerful to represent index domains, index expressions, constraints, and proofs. Representing constraints and proofs necessitates a framework such as LF (Harper et al., 1993) that involves binding and scope, which are used to model the consequence relation of a logic. Then indexed families of types can be defined and implemented by providing structural induction, modulo α-renaming, over the terms of the framework.

To illustrate these ideas, we define a type of queues whose contents are tracked by the type system. For tracking the contents of
queues, we require a small piece of finite set theory. An appropriate index domain and logic are defined in Figure 1. This LF signature specifies a two-sorted predicate logic consisting of a sort of individuals and a sort of finite sets of individuals. It provides operations for constructing finite sets, including the empty set (void), the singleton consisting of one individual, the union of two finite sets, and the difference of two sets. It also specifies a logic for reasoning about them, such as the LF types \( \text{prop} \) and \( \text{pf} \) above. The above example defined only atomic propositions (equality and inequality), but in general a programmer may wish to define connectives such as implication and quantification, encoded using variable binding in LF. Adequate higher-order representations of proofs rely crucially on the weakness of LF’s function space, which requires a distinction between the logical framework and the computational language.

To implement modules like the above, the programmer will need to define indexed families of types (such as \( \text{elt} \) and \( \text{queue} \)) and implement operations on them, which is supported by computation on indices. Computation permits:

1. Meta-theoretic reasoning about indices. For example, it may require non-trivial reasoning to discharge the index constraint \( \text{pf}(\text{neq}(s,\text{void})) \) when \( s \) is a set expression consisting of a series of unions and differences involving index variables.

2. Proof-producing run-time checks. For example, an operation \( \text{isEmpty} : \forall s : \text{set} \ (\exists t : \text{queue}[t] \ : \text{pf}(\text{eq}(t, \text{union}(s, \text{sing}(i)))) \) permits the safe discharge proof obligations, such as the premise of \( \text{deq} \) when the constraint is not statically provable, or the cost of proving it is deemed too high.

3. Computed index expressions. For example, an alternate representation of finite sets would provide a more limited collection of set constructors (singleton set, empty set, union) and implement the remaining operations (such as set difference) as functions from sets to sets. Computed indices are useful because their definitions are automatically expanded by the notion of definitional equality in the type system.

4. Run-time actions of index constraints. For example, for a particular application, a \( \text{queue}[s] \) might suffice for a \( \text{queue}[s’] \) whenever \( s’ \) is a subset of \( s \). This subtyping relationship can be expressed as a constraint \( \text{subset}(s’, s) \) and witnessed using a run-time induction over proofs to define a coercion on queues. Such a coercion might be the identity, if the implementation of queues builds in this cumulativity, or it may remove the extra elements in the \( \text{queue}[s] \), if the indexing tracks the queue contents precisely.

5. The definition of indexed types by induction on their indices. For example, the underlying implementation of \( \text{queue}[s] \) might be a type of arrays whose representation is defined as a function of the element type (Harper and Morrisett, 1995).

6. The implementation of such indexed types, which often require a corresponding run-time induction on the index.

**Contributions.** In this paper, we define a type theory which supports extensible indexed types. It is the first type theory to provide both (1) a logical framework for defining index data, and (2) computation over the terms of the framework, both at compile-time and at run-time. Following (Harper and Stone, 2000), our type theory is intended as a semantically proximate target for the elaborative semantics of an external language with indexed types, which we will consider in future work.

The central technical challenge in the design of this type theory is that the logical framework includes variable binding, so the
method for computing with it must provide recursion over higher-order data. To facilitate the study of this problem, we consider a more restricted logical framework than full LF. Specifically, we study a framework of abstract binding trees (Constable et al., 1986; Fiore et al., 1999; Pitts, 2006) that is expressive enough to represent higher-order abstract syntax but not judgements. We claim that this simplification retains the essential difficulties of the problem we need to solve, in that it requires inducting over higher-order data.

The technical contributions of our type theory are novel methods for integrating a logical framework into a programming language and computing with higher-order data. The terms of the framework are introduced into the programming language in a manner that is similar in spirit, but different in technical detail, to Pientka (2006)'s: the key idea is that the free variables of a framework term are bound locally by the introduction form, so no framework variables scope over programming language terms. This introduction form permits a direct transcription of the induction principle for the framework as a dependently typed recursor.

Our type theory has the following features:

**The Phase Distinction.** Our type theory makes a phase distinction (Harper et al., 1990) between static (compile-time) and dynamic (run-time) data, where static data is permitted to depend only on other static data, but dynamic data is permitted to depend on both static and dynamic data. The phase distinction is reflected in the syntax of our calculus, which is stratified into a static part of *constructors* classified by *kinds*, and a dynamic part of terms classified by *types*. Indices are construed as compile-time data, as they influence the static properties of a program, and are integrated into the constructor and kind level. We exploit the dependence of static data on static data to permit compile-time computation with indices, and we exploit the dependence of dynamic data on static data to permit run-time computation with indices.

**Modularity.** Modularity is the only way to manage the complexity of large software systems. Modularity is achieved by separating the implementation of a program component from its clients via an *interface*, which describes everything the clients are permitted to know about the implementation. Indexed types enrich the language of interfaces in a way that is compatible with modularity: what is known about an implementation is still exactly what is written in its type. In contrast, true dependency, where programs are used as their own models, is incompatible with modularity, as it requires that the very implementation of an operation be revealed in its interface.

**Extensibility.** As we argue with the examples below, our type theory is an extensible framework for the definition of and computation with programmer-defined indices. The programmer may define new index domains, indexed families of types, index-level functions, proof-manipulating run-time checks, and run-time proof actions, and he may reason about indices in the kind and constructor level of the programming language, which doubles as an adequate metalogic. In future work, we plan to build on the framework we develop here to support extensible decision procedures used to discharge proof obligations during type checking. When proofs are taken as fundamental, decision procedures can be construed as automated assistance for constructing omitted proof information, in a process analogous to type inference. This viewpoint has the important advantage that it is compatible with programmer-defined decision procedures, as the definition of the programming language itself does not depend on what automation is provided.

The remainder of this paper is organized as follows. In Section 2, we give an overview of abstract binding trees. In Section 3, we give an informal description of our techniques for integrating abstract binding trees into a programming language and supporting recursion over them. In Section 4, we illustrate several of the applications of computation with indices discussed above. In Section 5, we give a formal account of the judgements and metatheory of the calculus. In Section 6, we compare our work with the many other proposals for indexed programming and for inducting on higher-order data, and in Section 7, we discuss future work and conclude.

2. Overview of Abstract Binding Trees

2.1 Definition of the Framework

The logical framework that we will use in this paper, abstract binding trees, is a simple framework sufficient for representing syntax with binding; it is defined in Figure 3. An abstract binding tree (ABT) is a variable, an abstractor, or an operator applied to a spine, where a spine is a list of terms. An ABT is classified by a valence, which is a natural number describing the number of variables bound by the term. The judgement $\Gamma \vdash N : I$ defines this classification: A variable has valence zero, as long as it is declared in the context $\Psi$. The valence of an abstractor is one more than the valence of its body—e.g., $x. y. x$ has valence 2. An application of an operator has valence zero, provided the argument spine has the arity associated with the operator. An arity is a list of valences. A spine $S$ has arity $L$ if the spine is a list of terms of the valences specified by $L$, as defined by the judgement $\Gamma \vdash S : L$. These typing judgements are implicitly parametrized by a signature $\Omega$, associating arities with operators, which is invariant throughout a derivation. ABTs are considered up to $\alpha$-equivalence of the bound variables. For readability, we will often omit the final $\cdot$ and $\langle \rangle$ in non-empty arities and spines, writing, for example, $z \times z$ for $z \times z \times \cdots$.

As an example object language, we can represent the syntax of the untyped $\lambda$-calculus with the following signature $\Omega_S$:

\[
\begin{align*}
\lambda & : S z \Rightarrow z \\
\text{app} & : z \times z \Rightarrow z
\end{align*}
\]

That is, the operator app takes two arguments, neither of which bind any variables, and the operator lam takes one argument, which binds one variable. Terms of the untyped $\lambda$-calculus are encoded as in the following examples:

\[
\begin{align*}
x & \Rightarrow x \\
\langle x, y \rangle & \Rightarrow \text{app}(\langle x; y \rangle) \\
(\lambda x. x) y & \Rightarrow \text{app}(\langle \lambda x. (x; x); y \rangle)
\end{align*}
\]

Object-language variables are represented by framework variables using higher-order abstract-syntax, as in LF (Harper et al., 1993), naturally representing $\alpha$-equivalence classes of object-language terms.
2.2 Induction Principle

We can reason about ABTs using rule induction over the judgement $\Psi \vdash N : I$. Intuitively, the induction principle for this judgement is structural induction, modulo $\alpha$-conversion, over the terms of the framework. Formally, the hypothetical judgement $\Psi \vdash N : I$ is a simultaneous inductive definition of a family of categorical judgements, indexed by contexts $\Psi$, on ABTs $N$ and valences $I$. Thus, the induction principle for this judgement permits us to prove, all at once, a context-indexed family of propositions about an ABT $N$ and valences $I$, which we will write as $P_\Psi(I, N)$. We extend this notation to spines by writing $P_\Psi(L, S)$, where

$$P_\Psi(:, :) = true$$

$$P_\Psi(I \times L, N; S) = P_\Psi(I, N) \land P_\Psi(L, S)$$

Then the induction principle for ABTs is stated as follows:

**Definition 2.1: Induction Principle for $\Psi \vdash N : I$.**

1. For all $\Psi$, for all $x$ in $\Psi$, $P_\Psi(x, x)$.
2. For all $\Psi$, $I$, $N$, for some/any $x$ such that $x \not\in \Psi$, if $P_\Psi(x, I, N)$ then $P_\Psi(s I, x, N)$.
3. For all $\alpha : L \Rightarrow z$ in $\Omega$, for all $\Psi$ and $S$, if $P_\Psi(L, S)$ then $P_\Psi(z, o, S)$.

Then for all $\Psi$, for all $I$, for all $N$ such that $\Psi \vdash N : I$, $P_\Psi(I, N)$.

Intuitively, this induction principle says that to prove a property of an arbitrary ABT of an arbitrary valence in an arbitrary context, it suffices to give (1) a case for every variable, (2) a case for an abstractor, for some and hence any fresh bound variable, in terms of the inductive result on its body, and (3) a case for each operator in terms of the inductive results on its arguments.

2.3 Computational Content

The computational content of the above induction principle for ABTs is a recursion operation. For exposition, we first consider a simply-typed iteration operation. Assuming a type $\alpha$ of ABTs in a programming language, this iterator would have the following type:

$$\text{ABTiter}_\Omega : \forall \alpha. \{ \text{var} : \alpha, \text{abs} : \alpha \rightarrow \alpha, \text{ops} : \{ o_1 : \alpha o_1 \rightarrow \alpha, \ldots \} \} \rightarrow \alpha$$

where $\text{ops}$ has one entry for each operator in $\Omega$, and $\alpha o_1$ is a product $\alpha^n$ whose length $n$ is the number of arguments to $o$. Computationally, this iterator would analyze the provided ABT and defer to the appropriate case, supplying the inductive calls for the arguments to the cases.

For example, consider the function assigning a size to an untyped $\lambda$-term:

- $\text{size}(x) = 1$
- $\text{size}(\lambda x.e) = 1 + \text{size}(e)$
- $\text{size}(e_1 e_2) = 1 + \text{size}(e_1) + \text{size}(e_2)$

Using the above signature for $\lambda$-terms, we could implement this function as follows:

$$\text{size} : \alpha \rightarrow \text{nat} = \lambda x. \text{ABTiter} \{ \text{var} = 1, \text{abs} = \lambda x.x, \text{ops} = \{ \text{lam} = \lambda x.x+1, \text{app} = \lambda (x,y).x+y+1 \} \}$$

The abstractor case is the identity function because traversing an abstractor does not contribute to the above definition of size. To write more interesting functions over ABTs, such as those that compute ABTs as results, we must describe the way ABTs are introduced into the programming language.

3. Computing With Indices

We now give an informal description of our method for integrating ABTs into the programming language and providing computation with them. We focus first on the kind and constructor level of the language. To admit a precise kinding for recursion over ABTs, the kind and constructor language is dependently typed. In particular, function kinds $K_1 \rightarrow K_2$ are generalized to dependent function kinds $\Pi x : K_1, K_2$, though we use the former notation as a shorthand when the bound variable does not occur.

3.1 Introduction Forms for ABTs

To motivate the operations we provide on ABTs, consider what would be necessary to define an inductive identity function:

$$\text{id} : \alpha \rightarrow \alpha = \lambda x. \text{ABTiter} \{ \text{var} = \{ \text{the given variable}, \text{abs} = \lambda e. \text{ABTiter} \{ \text{abstract the free variable of } e, \text{ops} = \{ \text{lam} = \lambda e. \text{ABTiter} \{ \text{apply } \text{lam} \text{ to } e, \ldots \} \} \} \}$$

Completing this function requires constructing ABTs using variables from the programming language. In particular, in the abstractor case, the programming language variable $e$, representing the result of the recursive call, must stand for an ABT with an extra free variable $x$; to complete the case, we must return an $\alpha \times e$ that reabstracts this variable.

This case exposes the key difficulty of integrating the logical framework with the programming language: variables from the framework must, in some sense, be free in terms from the programming language. One approach to this problem, studied in work on nominal logic (Pitts, 2003), is to add a type of names to the programming language, so that the abstractor case of the induction principle provides both a name $x$ as well as the recursive result, in which $x$ is potentially free. However, because the programmer has access to the concrete bound name, it can require non-trivial proof to ensure that the result of the branch is independent of the particular choice of bound name, which is necessary to ensure that the computation respects $\alpha$-equivalence (Pitts and Gabbay, 2000; Pottier, 2007).

In this work, we take a different approach. First, an ABTs is introduced into the programming language using a syntactic construct that locally binds its free variables; no framework variables are free in programming language expressions. Second, a free variable of a computed ABT can be reabstracted using an explicit substitution. These mechanisms permit programming with open terms without giving concrete access to the free variables, which ensures syntactically that all computations respect $\alpha$-equivalence. Moreover, unlike nominal-logic-based approaches where reasoning about freshness permeates all of the rules of the type system, our method of adding a logical framework to a programming language is a modular extension.

To ensure that variables are used correctly, it is necessary to track the free variables of an ABT in its kind. To introduce ABTs, the syntax of constructors includes an injection of the syntax of ABTs $N$. This form is classified by the kind $\text{ABT}(L)$, where $L$ represents the context of free variables of the ABT, and $I$ is the valence of the ABT. The injection $N$ binds the free variables represented by the context $L$. This permits opening an abstractor $x. N : \text{ABT}(L)$ to term with a free variable, which we can informally write as $N : \text{ABT}(L, x : z)$, without binding the variable $x$ at the level of the programming language.

For technical reasons, we regard the free variables of the ABT as a product structure, reusing the syntax $L$ of arities, which we will
also refer to as contexts, to classify it. The programmer may refer to
free variables from the context \( L \) via projections. We extend the
syntax of ABT as follows:

\[
\begin{align*}
N & ::= \ldots | \pi \downarrow U \\
U & ::= \ldots | \text{it} \downarrow \pi \downarrow U
\end{align*}
\]

The construct \( \text{it} \) refers to the entire context, and projections \( \pi \downarrow U \) and \( \pi \downarrow \text{it} \) decompose it. For example, \( \text{app} (\pi \downarrow \text{it}; \pi \downarrow \text{it}) \) has kind \( \text{abt}_2(x \times z) \) and represents the application of the operator
app to the two free variables. An ABT \( N \) injected into the pro-
gramming language must be closed with respect to the ordinary
variable context \( \Psi \); all free variables must be written as pro-
jections from \( L \). However, locally bound variables in abstractors
are still bound in the usual manner. For example, the constructor
\( \text{lam} (x. \text{app} (x; \pi; \text{it})) \) has kind \( \text{abt}_4(z) \) and represents the \( \lambda \)-term
\( \lambda x. x y \), where \( y \) is a free variable.

The free variables of an ABT can be rebound by applying a
computed ABT of kind \( \text{abt}_1(L) \) to a substitution for the variables in
\( L \). This application is represented by one additional form of ABT:

\[
\begin{align*}
N & ::= \ldots \mid P \cdot S
\end{align*}
\]

which is the the application of a constructor \( P \) to a spine \( S \). Informally, \( P \cdot S \) has valence \( I \) if \( P : \text{abt}_1(L) \) and \( S : L \). The spine
\( S \) defines a (nameless) substitution for all of the free variables of \( P \).
Because the spine may mention locally bound variables, this form
can be used to bind an ABT with a free variable into an abstractor
without directly mentioning the free variable. As a specific instance of
this, the following code binds the one free variable in the ABT \( w \):

\[
\begin{align*}
u : \text{abt}_2(z) & \downarrow x. u \cdot x : \text{abt}_4(\cdot)
\end{align*}
\]

Below, we discuss the general case of abstracting one variable
in a term with arbitrary other free variables. Operationally, when
the constructor \( P \) is reduced to an actual ABT \( N \), the form \( P \cdot S \)
is reduced by replacing projections from \( P \) in \( N \) with the values supplied
by the spine.

### 3.2 Variable Contexts and Valences

The induction principle for ABTs (DEFINITION 2.1) quantifies
over valences and contexts. For example, the abstractor case of
the induction principle quantifies over a valence \( I \) and a context
\( \Psi \). Because valences and contexts are represented as data in the
programming language, the recursor must similarly quantify over
them. However, we have not yet introduced variable valences or
contexts. Thus, we add two new kinds, \( \text{vale} \) and \( \text{ctx} \), classifying
valences and contexts, along with injections \( I \) and \( L \) inhabiting them.
Additionally, as with ABTs, it is necessary to extend the
syntax of valences and contexts to permit computed forms:

\[
\begin{align*}
I & ::= \ldots | P \\
L & ::= \ldots \mid P \times L \\
S & ::= \text{spn}(U); S : P \times L
\end{align*}
\]

The context \( P \times L \) is well-formed if \( P \) has kind \( \text{ctx}; \) i.e., \( P \) stands
for an unknown part of the context. The only operation we provide
on such an unknown context is the ability to use it in a spine, where
it stands for the identity substitution on that context. This has the
following typing rule:

\[
\begin{align*}
U : P \times L' & \quad S : L \\
\text{spn}(U); S & : P \times L
\end{align*}
\]

For example, the following code applies the variable \( u \), which is an
ABT in variable context \( w \), to the identity spine for that context:

\[
w : \text{ctx}, u : \text{abt}_2(w \times \cdot) \vdash w \cdot (\text{spn}(\text{it}); ()) : \text{abt}_4(w \times \cdot)
\]

As a notational convenience, when \( L \) is \( P \times \cdot \), we simply write
it for \( \text{spn}(\text{it}); () \); for example, the above ABT constructor will be
written \( \text{w-it} \).

### 3.3 Structural Recursion over ABTs

Next, we internalize the induction principle as a recursor in the
usual manner of dependent type theory (Constable and Mendler,
1985; Constable et al., 1986; Luo, 1994). In each branch, the
recursor provides both the exposed subterm and the inductive result
on it. A recursor, rather than an iterator, is necessary to state the
fully dependent rule, as the type of the elimination form depends
on the term being eliminated.

In full generality, the \( \text{ABTRec} \) construct must be defined for any
ABT signature \( \Omega \), but for presentational reasons we first consider
the special case of \( \text{ABTRec} \) for the signature representing the
untyped \( \lambda \)-calculus that we discussed above. We present the rule
for the general case in Section 5.

Figure 4 contains the kinding rule for \( \text{ABTRec} \). While at first
glance this rule may seem complex, it is simply a transcription
into dependent type theory of the induction principle for ABTs
described above in DEFINITION 2.1. The result kind \( K \), which
corresponds to the proposition \( P \) in the induction principle, is
parametrized by a context, a valence, and an ABT in that context
and valence. Substitutions into the result kind \( K \) correspond to
the arguments of the property \( P \). To improve readability, we write
\( K [C_1] [C_2] [C_3] \) for \( K [C_1/\text{w0}] [C_2/\text{io}] [C_3/\text{a0}] \) for the three free
variables of \( K \). Some kinds of the inductive hypotheses and results
vary in each branch, following the proof obligations in the inductive
principle. This variation, which is standard in inductive family
elimination forms, propagates information by specializing the types
in each branch of the recursor.

The overall result kind in the conclusion of the rule is the
instantiation of \( K \) with the scrutinized ABT \( P \) and its valence
and context. The variable case \( C_1 \) covers an arbitrary variable
in an arbitrary context by assuming left and right contexts \( w \) and \( w' \)
and considering a variable between them. Up to associativity and
unification of products (for details, see Section 5), any variable has this
form. The abstractor case \( C_2 \) is a direct analogue of the abstractor
case of the induction principle: it assumes the inductive result for
an ABT of an arbitrary valence in an arbitrary context \( w \) plus one
distinguished free variable, and it covers the case for the abstractor
formed by binding this distinguished variable. The case \( C_3 \) for the
operator \( \lambda \) assumes an inductive result for one term of valence
\( z \) and must cover the case for \( \text{lam} \) applied to that term. Similarly,
the case \( C_4 \) for the operator \( \text{app} \) assumes a constructor-level pair
of inductive results of valence \( z \) and must cover the case for \( \text{app} \)
applied to them.

The computational behavior of \( \text{ABTRec} \) is straightforward: it
examines the scrutinized ABT and defers to the appropriate case
depending on its form, substituting \( \text{ABTRecs} \) on subterms for the
inductive variables.
We present two simple examples of how this recursor is used in Figure 5. First, we complete the inductive identity function example from the beginning of this section. In the variable case, projection from the context is used to return the distinguished variable. In the abstractor case, the spine application P.S is used to rebind the extra free variable, as we saw a special case of above. As another simple example, assuming a kind nat of natural numbers, we could write code to count the number of bound variables (abstractors) in an ABT. The abstractor case increments the count, the variable case returns 0, and the operator cases sum the inductive results.

### 3.4 Run-time Computation

To support run-time computation, we extend the run-time programs of the language with an elimination form for ABTs. Instantiated to the signature $\Omega_{\lambda}$, this elimination form has the following syntax:

\[
\text{ABTcase} [w_0, i_0, u_0, A](P, w, w', E_4, w, i, u, E_3, w, u, E_3, w, u, E_4)
\]

This construct is analogous to ABTRec, except each branch is a run-time term $E$ rather than a compile-time term $C$, and the result classifier is a family of types $A$ rather than kinds $K$. The eliminated ABT remains a constructor of kind $\text{abt}_i (L)$. Because the run-time language includes general recursion, a case-analysis construct, rather than a recursor, suffices, so the abstractor and operator branches do not bind variables standing for the recursive call. The typing and dynamic semantics of this construct are analogous to ABTRec.

### 3.5 Propositional Equality

An indexed family of types $\tau[i]$ defines a functional dependency between the index domain $I$ and the types $\tau[i]$—every index determines a type, and equal indices determine equal types. Type equality in turn influences type checking. In the presence of computed indices, some desired index equalities, and therefore type equalities, may not be directly derivable using the notion of definitional equality built into the type system. For example, for a programmer-defined index addition operation $\text{plus}$, it requires an inductive argument to show that the indices $\text{plus} i j$ and $\text{plus} j i$ are equal, and a decidable notion of definitional equality cannot perform arbitrary inductive reasoning. To support explicit proofs of such equalities, we include a notion of propositional equality, expressed as an indexed family of kinds $\text{Id}_{K,K}(C, C')$. This family of kinds is the heterogeneous propositional equality of McBride (2000); heterogeneous equality is useful in the presence of dependency, as it allows equations between constructors whose kinds are only propositionally equal. The introduction form is reflexivity, which proves $\text{Id}_{K,K}(C, C)$, and the elimination form for a proof of $\text{Id}_{K,K}(C, C')$ propagates the fact that $C$ and $C'$ are equal. As with ABTRec, we include elimination forms for equality at both the static and dynamic levels.

### 4. Examples

We now present examples of programming with indexed types in our calculus.

**Defining an Index Domain and Index Expressions**  
An index domain is defined by an ABT signature:

\[
\begin{align*}
\text{prod} : & \text{z x z} \Rightarrow \text{z} \\
\text{sum} : & \text{z x z} \Rightarrow \text{z} \\
\text{one} : & \Rightarrow \text{z} \\
\text{let} : & \text{z x s z} \Rightarrow \text{z}
\end{align*}
\]

This signature represents the types of a simple object language with products, sums, unit, and, to illustrate programming with binding, a lettype construct, as one finds in linguistic approaches to managing sharing (Petersen, 2005). A richer logical framework, such as LF, would permit the definition of index propositions and proofs as other constants in the signature.

Rather than formally instantiating the typing rule for ABTRec with this signature, we simply note the types of the operator cases: the cases for prod and sum have the same typing as the case for app above; the case for one binds two variables with kind unit (since the operator has no subterms); and the case for let binds a pair of subterms, one with valence z and the other with valence s z, and the corresponding inductive results. For readability, we label the operator cases with the syntax $o : w. u. r. C$.

**Computing Index Expressions: Normalization.**  
In some of the uses of this index domain below, it will be necessary to expand away the let bindings to see the normal form of the represented type. Normalization is defined as a constructor-level function from indices to indices. It uses an auxiliary function

\[
\text{subst} :: \text{ctx} \cdot \text{abt}_i[w] + \text{abt}_4[w] \Rightarrow \text{abt}_4[w]
\]

which substitutes the first ABT for the distinguished variable bound by the abstractor of the second. In an informal ML-like notation with pattern matching and recursive calls, normalization is defined as follows:

**fun norm w i u =**

\[
\text{ABTcase u of}
\]

\[
\begin{align*}
\text{var} (w, w') & \Rightarrow \pi_1 (\pi_2 \text{ it}) \\
\text{abs} (u : \text{abt}_1[z x w]) & \Rightarrow (x. (\text{norm } [z x w] u \cdot (x, \text{ it}))) \\
\text{one} & \Rightarrow \text{one} \cdot () \\
\text{prod} (x : \text{abt}_2[w], y : \text{abt}_4[w]) & \Rightarrow \text{prod} \cdot (\text{norm } w z x) \cdot \text{it}, (\text{norm } w z y) \cdot \text{it}) \\
\text{sum} (x : \text{abt}_2[w], y : \text{abt}_4[w]) & \Rightarrow \text{sum} \cdot (\text{norm } w z x) \cdot \text{it}, (\text{norm } w z y) \cdot \text{it}) \\
\text{let} (x, y) & \Rightarrow \text{subst} (x, y)
\end{align*}
\]

The let case uses the auxiliary substitution function, and every other case is compositional, reconstructing an ABT from the inductive results, just as in the identity function above. The formal definition in the syntax of our calculus is the following:
We also define a family of types \[\text{key} : \text{abt}_{[\cdot]} \rightarrow \text{type} = \Lambda u.\]
\[\text{ABTRec}_{[\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\·\]

Implementing Operations on Indexed Types: Tries. Implementing tries requires run-time computation with indices. For example, the lookup function is defined as follows (where, for space reasons, we elide some cases):

\[
\text{lookup} : \forall k : \text{abt}_{[\cdot]}(\cdot) k \rightarrow \text{Abt\_trie } k d \rightarrow (d+1) \\
\text{fun lookup } k = \\
\text{ABTcase}_{[\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\·\]

Run-time Actions and Indexed Datatypes. The run-time elimination form for identity may be used to retype a term on the basis of a proof of equality. One application of this is implementing indexed datatypes (i.e., inductive families (Constable and Mendler, 1985; Constable et al., 1986; Luo, 1994) or GADTs (Cheney and Hinze, 2003; Xi et al., 2003)), which are another common way of defining an indexed family of types.

For example, we might use the above index domain to index the representation of a programming language with types, so that only well-typed terms are representable. In a surface syntax, a small example of such a datatype is as follows:

\[
\text{datatype term}[a::\text{abt}_{[\cdot]}] = \\
\text{Pair} : \forall a,b::\text{abt}_{[\cdot]}(\cdot). \text{term}[a] \rightarrow \text{term}[b] \rightarrow \\
\text{term}[\text{prod} \cdot \{a \cdot \text{it}, b \cdot \text{it}\}] \\
\text{Fst} : \forall a,b::\text{abt}_{[\cdot]}, \text{term}[\text{prod} \cdot \{a \cdot \text{it}, b \cdot \text{it}\}] \rightarrow \text{term}[a] \\
\]

Note that the result types of the constructors vary in each branch.

It is well-known that indexed datatypes may be reduced to ordinary datatypes with equality constraints (Cheney and Hinze, 2003; Sheard, 2004; Sulzmann et al., 2007). We formulate this idea in terms of two very simple type-theoretic ingredients: iso-recursive constructors of higher kind (Crará and Weirich, 1999; Harper and Stone, 2000) and identity proofs. Iso-recursive constructors are introduced via a type \( \mu K \) \((u, C_1, C_2)\). The constructor \( C_1 \) morally has kind \((K \rightarrow \text{type}) \rightarrow (K \rightarrow \text{type}) \) (though in fact the variable \( u \) stands for the \( K \rightarrow \text{type} \) argument), and the type \( \mu K \) \((u, C_1, C_2)\) is the application of its fixed point to \( C_2 \). Term constructors roll and unroll witness the isomorphism between \( \mu K \) \((u, C_1, C_2)\) and \((C_2 \langle v.\text{mu}K(u, C_1, v)/u \rangle)/C_2\).

This type can model datatypes with varying results by using propositional equality to capture the constraints on the result type. For example, the above type \text{term} is defined to be the type:

\[
\text{norm} : \Pi w : \text{ctx}\_\Pi :: \text{val}\_\text{abt}_{[\cdot]}(\cdot) w \rightarrow \text{abt}_{[\cdot]}(\cdot) w = \\
\Lambda w : \text{ctx}\_\Pi.\text{ABTRec}[\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\·
\]
We write $N'$ for and $S'$ for shifting each $U$ in $N$ by one:

\[
\begin{align*}
\vdash (π_1 U) & \rightarrow \pi_1 (π_2 U) \\
\vdash (spn(U); S) & \rightarrow spn(π_2 U); S
\end{align*}
\]

else defined compositionally.

pivot($L$, $U$) decomposes $L$ as $L_1 \times \times L_2$ where $z$ corresponds to $π_1 U$:

\[
\begin{align*}
pivot(I \times L, iL) & = (i, L) \\
pivot(I \times L, π_2 U) & = (I \times L_1, L_2) & \text{if pivot}(I, L) = (L_1, L_2) \\
pivot(P \times L, π_2 U) & = (P \times L_1, L_2) & \text{if pivot}(I, L) = (L_1, L_2)
\end{align*}
\]

select($H, \Omega, o$) = $w.u.E$ looks up the branch for $o$ (definition elided).

Each step rule has an implicit premise

\[
\vdash A[L; ctx/w][I; vale](P/u) :: A_0 : \text{type}
\]

for the $I$ and $L$ in the kind resulting from the normalization of $P$.

\[
\begin{align*}
\vdash P \downarrow π_1 U : \text{abt}_2(L) & \vdash pivot(L, U) = (L_1, L_2) \\
\vdash P \downarrow x. N : \text{abt}_2(L) & \text{ABTcase}_2[w. u. v. A](P, w. w'. E_1, \ldots) \rightarrow E'_1[L_1; ctx][L_2; ctx] : A_0
\end{align*}
\]

\[
\begin{align*}
\vdash P \downarrow o.S : \text{abt}_2(L) & \text{select}(H, \Omega, o) = w.u.E \\
\vdash \text{ABTcase}_2[w. u. v. A](P, w. u. w'. E_1, \ldots) \rightarrow E'[L; ctx][\text{fromspine}(S); \text{kind}_2(L')] : A_0
\end{align*}
\]

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example.png}
\caption{Dynamic Semantics of ABTcase}
\end{figure}

included into synthesizable constructors with a kind annotation, written $C : K$. The checked and synthesizable terms have an analogous structure. The syntax of constructor ABTRec branches $B$ is necessary because the number of operator cases of the ABTRec construct varies with the signature $Ω$; the syntax of term branches $H$ plays a similar role for ABTcase.

\section*{Judgements}

The calculus is defined by the following judgements:

- Valence and ABT context formation are defined by the judgements $Γ \vdash I$ valence and $Γ ⊢ L$ context. The judgements $Γ; Ψ; L ⊢ N \equiv I$ and $Γ; Ψ; L ⊢ S \equiv L'$ and $Γ; L ⊢ U ⇒ L$ define the formation of ABTs, spines, and context projection. These judgements require the context $Γ$ for computed valences, contexts, and ABTs; the ABT variable context $Ψ$ for locally bound variables; and the variable context $L$ for free variables.

- The three judgements $Γ \vdash K$ kind, $Γ \vdash C \equiv K$, and $Γ \vdash P ⇒ K$ define kind and constructor formation. The judgement $Γ \vdash C \equiv K$ checks a constructor against a known kind, whereas the judgement $Γ \vdash P ⇒ K$ synthesizes a kind from a constructor. The judgements $Γ \vdash K \vdash K'$ and $Γ \vdash C \equiv C'$ : $K$ define kind and constructor normalization. For constructor normalization, $C$, $C'$, and $K$ are inputs, and $C'$ is the output normal form; kind normalization is similar.

- The judgements $Γ \vdash E \equiv A$ and $Γ \vdash R ⇒ A$ define the static semantics of terms. The modes are analogous to those of the constructor judgements.

- The judgements $E \Rightarrow E'$ and $R \Rightarrow E : A$ define the dynamic semantics of terms as a call-by-value transition system.

All of the judgements are implicitly parametrized by an ABT signature $Ω$, which is fixed throughout the program.

A central issue in the presentation of our type theory is managing the non-trivial computational equalities between classifiers (kinds and types), which, as we saw above, must influence kind and typing. In our calculus, these equalities are managed by

\begin{itemize}
\item The three judgements $Γ \vdash K$ kind, $Γ \vdash C \equiv K$, and $Γ \vdash P ⇒ K$ define kind and constructor formation.
\item The judgements $Γ \vdash K \vdash K'$ and $Γ \vdash C \equiv C'$ : $K$ define kind and constructor normalization.
\item For constructor normalization, $C$, $C'$, and $K$ are inputs, and $C'$ is the output normal form; kind normalization is similar.
\end{itemize}
Abstract Binding Trees:

<table>
<thead>
<tr>
<th>Terms:</th>
<th>Synth. Term</th>
<th>Term Branch</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valence</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$I$ :=</td>
<td>$z</td>
<td>s \ I</td>
<td>P$</td>
</tr>
<tr>
<td>Context</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L$ :=</td>
<td>$x</td>
<td>t</td>
<td>P \times L$</td>
</tr>
<tr>
<td>Term</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$N$ :=</td>
<td>$x \times x</td>
<td>N \circ S</td>
<td>\pi_I U</td>
</tr>
<tr>
<td>Spine</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S$ :=</td>
<td>$()</td>
<td>N</td>
<td>S</td>
</tr>
<tr>
<td>Free Var.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$U$ :=</td>
<td>$\pi_S U</td>
<td>\pi^U U$</td>
<td></td>
</tr>
<tr>
<td>Signature</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Omega$ :=</td>
<td>$\bullet</td>
<td>\Omega, o</td>
<td>L \Rightarrow z$</td>
</tr>
</tbody>
</table>

Kinds and Constructors:

<table>
<thead>
<tr>
<th>Kind $K$ :=</th>
<th>Synth. Cons. $P$ :=</th>
<th>Con. Branch $B$ :=</th>
<th>Constructor $A, C$ :=</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{type}</td>
<td>\Pi u.K_1, K_2</td>
<td>\Sigma u.K_1, K_2</td>
<td>\text{unit}$</td>
</tr>
<tr>
<td>$</td>
<td>\Sigma K_1, K_2</td>
<td>(C_1, C_2)$</td>
<td>$\Sigma u.K_1, K_2</td>
</tr>
<tr>
<td>$</td>
<td>\text{val}</td>
<td>\text{ctx}</td>
<td>\text{abt}(L)$</td>
</tr>
<tr>
<td>$</td>
<td>\text{val}</td>
<td>\text{ctx}</td>
<td>\text{abt}(L)$</td>
</tr>
<tr>
<td>$</td>
<td>\text{val}</td>
<td>\text{ctx}</td>
<td>\text{abt}(L)$</td>
</tr>
<tr>
<td>$</td>
<td>\text{val}</td>
<td>\text{ctx}</td>
<td>\text{abt}(L)$</td>
</tr>
<tr>
<td>$</td>
<td>\text{val}</td>
<td>\text{ctx}</td>
<td>\text{abt}(L)$</td>
</tr>
<tr>
<td>$</td>
<td>\text{val}</td>
<td>\text{ctx}</td>
<td>\text{abt}(L)$</td>
</tr>
<tr>
<td>$</td>
<td>\text{val}</td>
<td>\text{ctx}</td>
<td>\text{abt}(L)$</td>
</tr>
<tr>
<td>$</td>
<td>\text{val}</td>
<td>\text{ctx}</td>
<td>\text{abt}(L)$</td>
</tr>
<tr>
<td>$</td>
<td>\text{val}</td>
<td>\text{ctx}</td>
<td>\text{abt}(L)$</td>
</tr>
<tr>
<td>$</td>
<td>\text{val}</td>
<td>\text{ctx}</td>
<td>\text{abt}(L)$</td>
</tr>
<tr>
<td>$</td>
<td>\text{val}</td>
<td>\text{ctx}</td>
<td>\text{abt}(L)$</td>
</tr>
<tr>
<td>$</td>
<td>\text{val}</td>
<td>\text{ctx}</td>
<td>\text{abt}(L)$</td>
</tr>
<tr>
<td>$</td>
<td>\text{val}</td>
<td>\text{ctx}</td>
<td>\text{abt}(L)$</td>
</tr>
<tr>
<td>$</td>
<td>\text{val}</td>
<td>\text{ctx}</td>
<td>\text{abt}(L)$</td>
</tr>
</tbody>
</table>

**Figure 6. Full Syntax**

<table>
<thead>
<tr>
<th>$\Gamma \vdash I$ valence</th>
<th>$\Gamma \vdash z$ valence</th>
<th>$\Gamma \vdash P .: \text{vale}$</th>
<th>$\Gamma \vdash P .: \text{ctx}$</th>
<th>$\Gamma \vdash L$ context</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash I$ valence</td>
<td>$\Gamma \vdash z$ valence</td>
<td>$\Gamma \vdash P .: \text{vale}$</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
</tr>
<tr>
<td>$\Gamma \vdash z$ valence</td>
<td>$\Gamma \vdash s$ valence</td>
<td>$\Gamma \vdash P .: \text{vale}$</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
</tr>
<tr>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
</tr>
<tr>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
</tr>
<tr>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
</tr>
<tr>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
</tr>
<tr>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
</tr>
<tr>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
</tr>
<tr>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
</tr>
<tr>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
</tr>
<tr>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
</tr>
<tr>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
</tr>
<tr>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
</tr>
<tr>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
</tr>
<tr>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
</tr>
<tr>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
</tr>
<tr>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
</tr>
<tr>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
</tr>
<tr>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
</tr>
<tr>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
</tr>
<tr>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
</tr>
<tr>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
</tr>
<tr>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
</tr>
<tr>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
<td>$\Gamma \vdash L$ context</td>
<td>$\Gamma \vdash P .: \text{ctx}$</td>
</tr>
</tbody>
</table>

**Figure 7. Selected Kind and Constructor Formation Rules**

- $\text{kind}_{\times}() = \text{unit}$
- $\text{kind}_{\times}(I \times L) = abt_{\times}(L') \times \text{kind}_{\times}(L)$
- $\text{ih}_{u.\Sigma.K_1.K_2}() = \text{unit}$
- $\text{ih}_{u.\Sigma.K_1.K_2}(P, I \times L) = \text{unit}$
- $\text{ih}_{u.\Sigma.K_1.K_2}(P, I \times L) = \text{unit}$
- $\text{tospine}(P, I \times L) = (\text{fst} P . \text{spn}(it)) ; \text{tospine}(\text{snd} P, L)$
- $\text{fromspine}(\cdot) = \{\}$
- $\text{fromspine}(N, L) = \{N, \text{fromspine}(L)\}$

**Figure 8. Auxiliary Operations**
keeping all classifiers in a normal form, and equality is defined to be syntactic identity of normal forms. To this end, we maintain several invariants: the context $\Gamma$ is only permitted to contain assumptions $u : K$ and $x : A$ for normal classifiers $K$ and $A$; the judgement $\Gamma \vDash P \Rightarrow K$ synthesizes a normal kind, whereas the judgement $\Gamma \vDash C \Leftarrow K$ presupposes that the given kind is normal; the term and ABT formation judgements maintain analogous invariants. Many of the rules contain normalization premises in order to maintain these invariants. One advantage of this style of presentation is that the typing rules are syntax-directed, which simplifies some aspects of the calculi’s metatheory.

**Formation Judgements.** In Figures 7 and 9, we present selected formation rules for the static part of the programming language. We show the full formation rules for the ABT level, as well as abridged kind and constructor rules. The rules we omit are straightforward adaptations of the standard bidirectional rules to maintain our normalization invariants. The ABT formation rules formalize what we described in the intuitive discussion above. For the rules about context formation permit variable contexts, but only permit the valence $z$, as all variables have valence $z$. The rule for $\pi^0_2 U$ ensures that the annotation $P$ matches the synthesized (and therefore normal) constructor $P'$ in the classifier using normalization.

Kind formation is as described above; observe that the rule for $\text{lId}_{K_1, K_2}(C_1, C_2)$ admits equations between constructors of different kinds. Among the rules for the synthesized constructors, the elimination form for identity may merit some discussion. The constructor $\text{lId}$ is annotated with a result kind pattern $v.p.K$. When $P$ proves $\text{lId}_{K_1, K_2}(C_1, C_2)$, the branch is checked with $C_1$ substituted for $v$, but the result kind is the substitution of $C_2$. Informally, this propagates the equality by permitting a $C_1$ to be used where a $C_2$ is expected. See McBride (2000) for further discussion of this elimination form for equality. The remaining rules in this figure define constructor checking; observe that the reflxity rule ensures that the constructors are identical, and that the premise of the the introduction form for ABTs $N$ refers to the judgement $\Gamma; \Psi; L \vdash N \Leftarrow I$ with an empty context $\Psi$, ensuring that all free variables of $N$ are represented as projections from $L$.

In Figure 9, we return to constructor synthesis with the rule for ABTRec, which is a slight variation on the rule presented earlier in Figure 4. The differences are as follows: For preciseness, we write out the empty spines and contexts $()$ and $\cdot$, rather than employing our convention of eliding them. Next, there are extra premises normalizing the substitutions into the pattern kind $K$; this is necessary to preserve the invariants of the judgements. Note that some of the substituted constructors are annotated with their kinds, which is necessary because variables stand for synthesizable constructors $P$. Finally, the rule refers to an auxiliary judgement to check the operator branches $B$, which vary with the signature $\Omega$.

The auxiliary judgement is also defined in Figure 9, and it employs three of the auxiliary operations defined in Figure 8. The judgement ensures that there is one branch for each operation in $\Omega$, and that each branch has the appropriate type. The operation $\text{kind}_{P,L}$ maps the arity of the constructor into a product kind of ABTs of the correct valence; it is used to compute the kind of the exposed subterms. The operation $\text{toSpine}(P, L)$ maps the constructor-level product $P$ to a spine that is used to state the result kind of the branch.

One subtlety of the ABTRec rule is that the variable case requires considering ABTs up to the associativity and unit laws on the product structure representing the context: it is only up to associativity and unit that every free variable is the middle variable in a context $w \times z \times w' \times \cdot$ for some contexts $w$ and $w'$. This list-like syntax of the contexts $L$ and spines $S$ is used as a canonical representative the associativity/unit equivalence classes.

We omit the typing judgements $\Gamma \vdash E \Leftarrow A$ and $\Gamma \vdash R \Rightarrow A$. Each rule defining these judgements is either a straightforward adaptation of the standard rule to the bidirectional setting, or, for $\text{lId}$ and ABTRec, a direct analogue of the constructor-level rule.

**Normalization.** Due to space restrictions, we do not present the rules defining our normalization algorithm. The algorithm is based on Harper and Pfenning (2005) and our previous extension to inductive types (Licata and Harper, 2005). It is kind-directed and computes long $\beta\eta$-normals for the kinds $\Pi w : K_1, K_2, \Sigma w : K_1, K_2$, and unit, and $\beta$-normal forms for $\text{Id}_{K_1, K_2}(C_1, C_2)$ and $\text{abt}(L)$.

Additionally, the algorithm reduces computed ABTs, valences and contexts (equating, for example, $s \ (s \ i : \text{vale})$ with $s \ (s \ i)$). The algorithm maintains the product structure $L$ representing the ABT context in our chosen list-like representation. For example, the context $\text{ctx}(z \times z) : \text{ctx} \ x \ z$ is normalized to the context $z \times (z \times \cdot)$. Because reassociating the products changes...
the meaning of the projections, the normalization algorithm must rewrite them; the annotation on $\pi^\beta_2$ $U$ provides sufficient information to perform this rewriting.

**Dynamic Semantics** The dynamic semantics are a standard call-by-value operational semantics, adapted slightly to account for the bidirectional syntax. In the auxiliary judgement $\Rightarrow A : A$, a synthesizing term steps to a checked term $E \Rightarrow E$ by-value operational semantics, adapted slightly to account for the $\beta$-rules for ABT-case in Figure 10. In the rules, we use the positional substitution notation $E[C]$ when the variable is clear from context. Note that the rules use two of the operations defined in Figure 8. The interested reader is referred to the online appendix for the remaining rules.

**Metatheory**

Type safety is stated as follows:

**Conjecture 5.1.** Assume $\Gamma \vdash A \equiv\Gamma$, and $\Gamma \vdash A \equiv A$, then $\Gamma \vdash A$.

**Preservation** If $E \Rightarrow E'$ and $\Gamma \vdash E \equiv A$ then $\Gamma \vdash E' \equiv A$.

**Progress** If $\Gamma \vdash E \equiv A$ then $E \Rightarrow E'$ or $E \equiv A$.

Because we have presented the type system algorithmically, many of the lemmas necessary for type safety, such as the inversion lemmas necessary for preservation and the canonical forms lemmas necessary for progress, are quite straightforward. The hard part of the proof is establishing properties of the normalization algorithm—e.g., that it commutes with substitution, that it is idempotent, and that every constructor has a unique normal form. We have reduced the type safety of the language to properties of normalization, but we label type safety as a conjecture because we have not yet checked every detail of the properties of normalization. Our proof is an adaptation of the logical relations method developed by Harper and Pfenning (2005), which we have applied to inductive types in previous work (Licata and Harper, 2005).

6. **Related Work**

Our work builds on a decade of research on integrating various forms of dependent and indexed types into practical programming languages and their implementations (Augustsson, 1998; Chen and Xi, 2005; Cheney and Hinze, 2003; Chin et al., 2005; Condit et al., 2007; Dunfield and Pfenning, 2004; Flanagan, 2006; Fogarty et al., 2007; Licata and Harper, 2005; McBride and McKinna, 2005; Nanevski et al., 2005; Peyton Jones et al., 2006; Sarkar, 2005; Shao et al., 2005; Sheard, 2004; Westbrook et al., 2005; Xi and Pfenning, 1998; Xi et al., 2003; Zenger, 1998). Relative to these designs, our type theory is the first to provide both a logical framework designed for representing higher-order index data as well as computation over the terms of the framework at both the static and dynamic levels. Many of these languages provide either computation (e.g., this is standard in the dependent type theories) or a logical framework but not both. Sarkar’s language is the most closely related work, as he used an ML-like language with types indexed by LF terms to implement a certified type checker. Relative to his work, our contribution is to point out that the LF/ML combination has many more uses than just that, and to extend the framework with recursion over indices. Permitting computation over indices is more general than precluding it, as we can always decide not to use the facility in a given setting. Our work also generalizes intensional type analysis (Crary and Weirich, 1999; Harper and Morrisett, 1995; Weirich, 2002), as the types of a language may be represented as a particular index domain.

Another broad category of related work concerns functional computation with higher-order data. Pientka (2006); Schürmann et al. (2001); Schürmann et al. (2005) present approaches to inducting over higher-order abstract syntax. All three of these papers consider either simply-typed or full LF; however, none consider a dependently typed computational language, as we do here. Schürmann et al. (2001) consider induction over the terms of a modal type within the computational language, whereas we distinguish the logical framework as a separate level, which provides greater freedom in the design of each. Both Schürmann et al. (2005) and Pientka (2006) employ a non-deterministic operational semantics that admits pattern matching failure and general recursion, which would be problematic in the kind and constructor level of our type theory. Our approach is quite similar in spirit to Pientka (2006)’s, as both are inspired by contextual modal type theory (Nanevski et al., 2007). However, Pientka follows the formalism of CMTT much more closely. This results in technical differences in how framework contexts, including variable contexts, are handled, and in what computed terms are permitted (we permit arbitrary constructors in the form $P.S$, rather than just variables), and in what may be abstracted (we permit variable valences but not substitutions, whereas Pientka permits variable substitutions but not framework types).

Nominal logic (Pitts, 2003, 2006) provides another approach to inducting on data with name binding. In the nominal induction principle, the case for an abstraction is proved from some suitable fresh name, which, by equivariance, shows that it holds for any such name. When this idea is integrated into a programming language, the entire type system must track supports in order to ensure that computations are equivariant, and therefore respect $\alpha$-equivalence (Pitts and Gabbay, 2000; Pottier, 2007). In contrast, our method of integrating and computing with a logical framework has no global effects on the type system.

7. **Conclusion**

In this paper, we have presented an extensible theory of indexed types. The essential ingredients of this theory are a logical framework, which permits the definition of index domains, and computation with the terms of this framework, which enables a variety of techniques necessary for programming with and reasoning about indices. We have shown how to integrate the logical framework into the static level of a programming language, and we have presented a novel method of computing with higher-order index data. We have formalized these ideas in an elegant type theory, suitable for use as a semantically proximate target for the elaboration of a more convenient surface language.

We have explored several technical extensions to the machinery described in this paper that are not presented here. One is to permit induction over valences, which we have used for writing the result pattern kinds of $ABTRec$ and $IdRec$ in some examples. Another is an extension of $ABTRec$ with one case for each free variable in the scrutinized ABT (e.g., when scrutinizing a $abt(L)$, the recur- sor would have one case for each variable in $L$), maintaining the general variable case to cover those variables that arise via induction. This extension permits the definition of functions that have different behavior on different free variables, such as substitution, or testing whether a designated variable occurs.

This paper is the first step in a long-term effort to realize the expressiveness of indexed types in a practical programming language. Our next step is to scale the methods presented here to the full LF logical framework. This paper shows how to induct over data with binding; the remaining challenge in handling LF is managing the types, which involve dependency, and which in turn necessitate considering induction in specified sets of contexts called worlds (Schürmann and Pfenning, 2003). Next, we plan to consider the extension of the calculus presented in this paper with an ML-style module system. Because our calculus is based on indexing
and not dependency, we believe that the module system will be a straightforward extension, modulo the technical issues of handling both induction over indices and singleton kinds. Finally, we plan to design a practical external language by elaboration. An important ingredient in this external language will be supporting programmer-defined decision procedures that can be used at compile-time to discharge proof obligations. The framework we have defined here is a foundation for such decision procedures: taking proofs as the definition of truth permits variation in decision procedures within a single language; computation with indices may be useful for programming such decision procedures; and programmer-defined index logics may create new opportunities for automation.

References