An Extensible Theory of Indexed Types

Daniel R. Licata   Robert Harper
Carnegie Mellon University
{drl,rwh}@cs.cmu.edu

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:

```
nil  : list[0]
cons : ∀n:nat.elt -> list[n] -> list[n+1]
```

Examples of indexing include dependent types, where the indices are the 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:

```
check : ∀e.∀t.term[e] -> typ[t]
          -> (∃d::of e t.unit) + unit
```

When this function returns true, it also returns a certificate, represented as an LF derivation of the judgement `of 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

```
signature FIN_SET_THY = sig
  fam ind : Type            % elements of sets
  fam set : Type            % finite sets

  obj void : set.
  obj sing : ind → set.
  objs union, diff : set → set → set.

  fam prop : Type           % propositions
  objs eq, neq : set → set → prop.

  fam pf : prop → Type      % proofs
  ...
end
```

Figure 1. LF Signature for Finite Set Theory

```
signature QUEUE = sig
  import Sets : FIN_SET_THY

  % elements indexed by ind
  typ elt : ind ⇒ type

  % queues indexed by sets
  typ queue : set ⇒ type

  val empty : queue[void]
  val enq :
   ∀ i:ind ∀ s:set elt[i] → queue[s] →
    ∃ t:set ∃ _:pf(eq(t,union(s,sing(i))))
     queue[t]
  val deq :
   ∀ s:set ∀ _:pf(neq(s,void)) queue[s] →
    ∃ i:ind ∃ t:set
     ∃ _:pf(eq(t,diff(s,sing(i))))
      elt[i] × queue[t]
end
```

Figure 2. Signature for Queues Indexed by Their Contents

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 finite sets. This consists of an LF type of propositions, including assertions of equality and inequality of finite sets. Crucially, it also consists of an LF type of proofs of propositions, as well as inference rules for generating such proofs (which we have elided).

Using these indices, we define a signature for run-time queue values in Figure 2. The elements of the queue are represented by a family of types, called `elt`, indexed by the LF type `ind` of individuals: there is a type `elt[i]` for every individual `i`, and different instances of the family may classify different terms. Next, the signature defines a type of queues, indexed by the LF type `set` of sets. The signature defines three operations on queues, using the indices to state how these operations act on the queue's contents. For example, the dequeue operation `deq` consumes a queue with non-void contents and produces an element along with a queue with that element removed.

In another implementation of queues, a programmer might wish to track only whether the queue is empty or not; in a third implementation, a programmer might wish to track the order of elements in a priority queue. LF permits natural representations of both first-order index domains such as natural numbers and sets, as well as index domains that require variable binding, such as the syntax of a programming language. Types indexed by abstract syntax have been used, for example, to implement type-safe interpreters by indexing the datatype of a programming language with a representation of the language's types (Pasalic, 2004; Peyton Jones et al., 2006).

Moreover, LF permits the definition of not just index data domains, but also the propositions and proofs used for reasoning about them, such as the LF types `prop` and `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 `elt` and `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 `pf(neq(s,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

   ```
   val isEmpty :
      ∀s:set (∃_:pf(neq(s,void)).unit) + unit
   ```

   permits the safe discharge proof obligations, such as the premise of `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 `queue[s]` might suffice for a `queue[s']` whenever `s'` is a subset of `s`. This subtyping relationship can be expressed as a constraint `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 `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 `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.

$$\begin{array}{llcl}
\textbf{Valence} & I & ::= & \mathsf{z} \mid \mathsf{s}\, I \\
\textbf{Arity} & L & ::= & \cdot \mid I \times L \\
\textbf{ABT} & N & ::= & x \mid x.\, N \mid \mathsf{o}{\cdot}S \\
\textbf{Spine} & S & ::= & () \mid N; S \\
 & & & \\
\textbf{Signature} & \Omega & ::= & \cdot \mid \Omega,\, \mathsf{o} : L \Rightarrow \mathsf{z} \\
\textbf{Context} & \Psi & ::= & \cdot \mid \Psi, x
\end{array}$$

$$\boxed{\Psi \vdash N : I} \qquad \frac{x \text{ in } \Psi}{\Psi \vdash x : \mathsf{z}} \qquad \frac{\Psi, x \vdash N : I}{\Psi \vdash x.\, N : \mathsf{s}\, I} \qquad \frac{\mathsf{o} : L \Rightarrow \mathsf{z} \text{ in } \Omega \quad \Psi \vdash S : L}{\Psi \vdash \mathsf{o}{\cdot}S : \mathsf{z}}$$

$$\boxed{\Psi \vdash S : L} \qquad \frac{}{\Psi \vdash () : \cdot} \qquad \frac{\Psi \vdash N : I \quad \Psi \vdash S : L}{\Psi \vdash N; S : I \times L}$$

Figure 3. Definition of Abstract Binding Trees

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 (x), an abstractor ($x.\, N$), or an operator applied to a spine ($\mathsf{o}{\cdot}S$), 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 $\Psi \vdash N : I$ defines this classification: A variable has valence zero, as long as it is declared in the context Ψ. 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 $\Psi \vdash S : L$. These typing judgements are implicitly parametrized by a signature Ω, associating arities with operators, which is invariant throughout a derivation. ABTs are considered up to α-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, $\mathsf{z} \times \mathsf{z}$ for $\mathsf{z} \times \mathsf{z} \times \cdot$.

As an example object language, we can represent the syntax of the untyped λ-calculus with the following signature Ω_λ:

$$\begin{array}{l}
\mathsf{lam} : \mathsf{s}\,\mathsf{z} \Rightarrow \mathsf{z} \\
\mathsf{app} : \mathsf{z} \times \mathsf{z} \Rightarrow \mathsf{z}
\end{array}$$

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 λ-calculus are encoded as in the following examples:

$$\begin{array}{rcl}
\mathsf{x} & \gg & x \\
\mathsf{x}\,\mathsf{y} & \gg & \mathsf{app}{\cdot}(x; y) \\
(\lambda\, \mathsf{x}.\, \mathsf{x})\, \mathsf{y} & \gg & \mathsf{app}{\cdot}(\mathsf{lam}{\cdot}(x.\, x); y)
\end{array}$$

Object-language variables are represented by framework variables using higher-order abstract-syntax, as in LF (Harper et al., 1993), naturally representing α-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 α-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 Ψ, 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 ABTs 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

$$\begin{array}{rcl} P_\Psi(\cdot, ()) & = & \textit{true} \\ P_\Psi(I \times L, N; S) & = & P_\Psi(I, N) \textit{ and } P_\Psi(L, S) \end{array}$$

Then the induction principle for ABTs is stated as follows::

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

1. For all Ψ, for all x in Ψ, $P_\Psi(\mathsf{z}, x)$
2. For all Ψ, I, N, for some/any x such that $x \notin \Psi$, if $P_{\Psi,x}(I, N)$ then $P_\Psi(\mathsf{s}\, I, x.\, N)$.
3. For all $\mathsf{o} : L \Rightarrow \mathsf{z}$ in Ω, for all Ψ and S, if $P_\Psi(L, S)$ then $P_\Psi(\mathsf{z}, \mathsf{o} \cdot S)$.

then for all Ψ, 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 any 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 ABT of ABTs in a programming language, this iterator would have the following type:

```
ABTiterΩ : ∀ α. {var : α,
                 abs : α  →  α,
                 ops : { o₁ : α_o₁  →  α, ... } }
                 -> abt -> α
```

where ops has one entry for each operator in Ω, and α_o is a product α^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 λ-term:

$$\begin{array}{rcl} \mathit{size}(\mathtt{x}) & = & 1 \\ \mathit{size}(\lambda\, \mathtt{x}.\, \mathtt{e}) & = & 1 + \mathit{size}(\mathtt{e}) \\ \mathit{size}(\mathtt{e}_1\, \mathtt{e}_2) & = & 1 + \mathit{size}(\mathtt{e}_1) + \mathit{size}(\mathtt{e}_2) \end{array}$$

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

```
size : abt -> nat =
λx. ABTiter {var = 1,
             abs = λx.x,
             ops = { lam = λx.x+1,
                     app = λ(x,y).x+y+1}
             } x
```

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 $\Pi\, u{::}K_1.\, K_2$, 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:

```
id : abt -> abt =
 λx. ABTiter
 {var=[the given variable],
  abs=λe:abt.[abstract the free variable of e],
  ops={lam=λe:abt.[apply lam to e], ... }
 } x
```

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 ABT x.e that reabstracts this variable.

This case exposes the key difficultly 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 α-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 α-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 $\mathsf{abt}_I(L)$, 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 $x.\, N : \mathsf{abt}_{\mathsf{s}\, I}(L)$ to term with a free variable, which we can informally write as $N : \mathsf{abt}_I(L, x : \mathsf{z})$, 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:

$$\begin{array}{lll} N & ::= & \ldots \mid \pi_1\ U \\ U & ::= & \mathsf{it} \mid \pi_2\ U \end{array}$$

The construct 'it' refers to the entire context, and projections $\pi_1\ U$ and $\pi_2\ U$ decompose it. For example, $\mathsf{app}{\cdot}(\pi_1\ \mathsf{it}; \pi_1\ \pi_2\ \mathsf{it})$ has kind $\mathsf{abt_z}(\mathsf{z} \times \mathsf{z})$ and represents the application of the operator app to the two free variables. An ABT N injected into the programming language must be closed with respect to the ordinary variable context Ψ; all free variables must be written as projections from L. However, locally bound variables in abstractors are still bound in the usual manner. For example, the constructor $\mathsf{lam}{\cdot}(x.\, \mathsf{app}{\cdot}(x; \pi_1\ \mathsf{it}))$ has kind $\mathsf{abt_z}(\mathsf{z})$ and represents the λ-term $\lambda\, \mathtt{x}.\, \mathtt{x}\ \mathtt{y}$, where $\mathtt{y}$ is a free variable.

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

$$N \quad ::= \quad \ldots \mid P{\cdot}S$$

which is the the application of a constructor P to a spine S. Informally, $P{\cdot}S$ has valence I if $P : \mathsf{abt}_I(L)$ 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 u:

$$u : \mathsf{abt_z}(\mathsf{z}) \vdash x.\, u{\cdot}x : \mathsf{abt_{s\,z}}(\cdot)$$

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 it 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 Ψ. 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, vale and 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:

$$\begin{array}{lll} I & ::= & \ldots \mid P \\ L & ::= & \ldots \mid P \times L \\ S & ::= & \ldots \mid \mathsf{spn}(U); S \end{array}$$

The context $P \times L$ is well-formed if P has kind 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:

$$\frac{U : P \times L' \quad S : L}{\mathsf{spn}(U); S : P \times L}$$

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

$$w : \mathsf{ctx}, u : \mathsf{abt_z}(w \times \cdot) \vdash u{\cdot}(\mathsf{spn}(\mathsf{it}); ()) : \mathsf{abt_z}(w \times \cdot)$$

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

```
id :: Πw::ctx.Πi::vale.abt_i[w] -> abt_i[w]
= λw.λi.λu.
    ABTRec [w0.i0._.abt_i0(w0)] (u,
         _._. π1 (π2 it),
         _._._.v. (x. v · (x ; it)),
         _.v. app · (fst v · it; snd v · it),
         _.v. lam · (v · it))


numBound :: Πw::ctx.Πi::vale.abt_i[w] -> nat
= λw.λi.λu.
    ABTRec [_._._.nat] (u,
         w.w'. 0,
         w.i.u.v. 1 + v,
         u.v. v,
         u.v. fst v + snd v)
```

Figure 5. Examples of ABTRec

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 ABTRec construct must be defined for any ABT signature Ω, but for presentational reasons we first consider the special case of ABTRec for the signature representing the untyped λ-calculus that we discussed above. We present the rule for the general case in Section 5.

Figure 4 contains the kinding rule for 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 $K[C_1/w_0][C_2/i_0][C_3/u_0]$ for the three free variables of K. The kinds of the inductive hypotheses and results vary in each branch, following the proof obligations in the induction 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 unit 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 lam assumes an inductive result for one term of valence $\mathsf{s\,z}$ and must cover the case for lam applied to that term. Similarly, the case C_4 for the operator app assumes a constructor-level pair of inductive results of valence $\mathsf{z}$and must cover the case for app applied to them.

The computational behavior of ABTRec is straightforward: it examines the scrutinized ABT and defers to the appropriate case depending on its form, substituting ABTRecs on subterms for the inductive variables.

$$\frac{\begin{array}{l}\Gamma, w_0 : \mathsf{ctx}, i_0 : \mathsf{vale}, u_0 : \mathsf{abt}_{i_0}(w_0) \vdash K\ \mathsf{kind} \\ \Gamma \vdash P : \mathsf{abt}_I(L) \\ \Gamma, w : \mathsf{ctx}, w' : \mathsf{ctx} \vdash C_1 : K[w \times \mathsf{z} \times w'][\mathsf{z}][\pi_1\ \pi_2\ \mathsf{it}] \\ \Gamma, w : \mathsf{ctx}, i : \mathsf{vale}, u : \mathsf{abt}_i(\mathsf{z} \times w), v : K[\mathsf{z} \times w][i][u] \vdash C_2 : K[w][\mathsf{s}\ i][x.\ u{\cdot}(x; \mathsf{it})] \\ \Gamma, w : \mathsf{ctx}, u : \mathsf{abt}_{\mathsf{s}\,\mathsf{z}}(w), v : K[w][\mathsf{s}\,\mathsf{z}][u] \vdash C_3 : K[w][\mathsf{z}][\mathsf{lam}{\cdot}(u{\cdot}\mathsf{it})] \\ \Gamma, w : \mathsf{ctx}, u : (\mathsf{abt}_{\mathsf{z}}(w) \times \mathsf{abt}_{\mathsf{z}}(w)), v : (K[w][\mathsf{z}][\mathsf{fst}\ u] \times K[w][\mathsf{z}][\mathsf{snd}\ u]) \vdash C_4 : K[w][\mathsf{z}][\mathsf{app}{\cdot}(\mathsf{fst}\ u{\cdot}\mathsf{it}; \mathsf{snd}\ u{\cdot}\mathsf{it})]\end{array}}{\Gamma \vdash \mathsf{ABTRec}[w_0.i_0.u_0.K](P, w.w'.C_1, w.i.u.v.C_2, w.u.v.C_3, w.u.v.C_4) : K[L][I][P]}$$

Figure 4. ABTRec for Ω_λ

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 $P{\cdot}S$ 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 Ω_λ, this elimination form has the following syntax:

$$\mathsf{ABTcase}[w_0.i_0.u_0.A](P, w.w'.E_1, w.i.u.E_2, w.u.E_3.,w.u.E_4)$$

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 $\mathsf{abt}_I(L)$. 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 $(\tau[i])_{i \in I}$ 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 in to the type system. For example, for a programmer-defined index addition operation *plus*, it requires an inductive argument to show that the indices *plus i j* and *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 $\mathsf{Id}_{K,K'}(C, C')$. 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 $\mathsf{Id}_{K,K}(C, C)$, and the elimination form for a proof of $\mathsf{Id}_{K,K}(C, C')$ 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:

$$\begin{array}{l}\mathsf{prod} : \mathsf{z} \times \mathsf{z} \Rightarrow \mathsf{z} \\ \mathsf{sum} : \mathsf{z} \times \mathsf{z} \Rightarrow \mathsf{z} \\ \mathsf{one} : \cdot \Rightarrow \mathsf{z} \\ \mathsf{let} : \mathsf{z} \times \mathsf{s}\,\mathsf{z} \Rightarrow \mathsf{z}\end{array}$$

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

```
subst::Πw::ctx.abt_z[w] * abt_s z[w] -> abt_z[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:

```
norm::Πw::ctx.Πi::val.abt_i[w] -> abt_i[w]
fun norm w i u =
  ABTcase u of
    var(w,w') => π1 (π2 it)
  | abs(u :: abt_i[z × w]) =>
     (x. (norm i [z × w] u) · (x, it))
  | one => one · ()
  | prod (x : abt_z[w], y : abt_z[w]) =>
     prod · ((norm w z x) · it, (norm w z y) · it)
  | sum (x : abt_z[w], y : abt_z[w]) =>
     sum · ((norm w z x) · it, (norm w z y) · it)
  | let (x, y) => subst (x,y)
```

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:

```
norm::Πw::ctx.Πi::val.abt_i[w] -> abt_i[w] =
λw.λi.λu.
 ABTRec[w.i._.abt_i[w]] (u,
   _._. π1 (π2 it),
   _._._._. (x. v · (x, it)),
   one  : _._._. one · ()
   prod : _._.v. prod · (fst v · it,
                         snd v · it)),
   sum  : _._.v. sum · (fst v · it,
                         snd v · it)),
   let  : _._.v. subst v)
```

Substitution is defined as follows:

```
open::Πw::ctx.abt_{s z}[w] -> abt_z[z × w] =
λw.λu. ABTRec[_._._.abt_z[z × w]] (u,
   _._.u._. u,
   else => π1 it)

subst::Πw::ctx.abt_z[w] * abt_{s z}[w] -> abt_z[w]
= λw.λp. (open w (snd p)) · ((fst p · it); it))
```

The function open passes from an abstractor with valence one to an ABT in an extended context with valence zero: in the abstractor case, it returns the exposed subterm. All remaining cases are impossible by kinding, as there is no other way to construct an ABT with valence higher than zero. However, when programming in a total language, some result must be specified. One possibility is to prove the impossibility of the cases (McBride (2000) describes idioms for doing so); another alternative, when desired result kind is inhabited, is to simply return some default value, as we do here (we use else to write a catch-all clause rather than writing each case explicitly).

Substitution consumes a pair of an ABT and an abstractor and returns the result of substituting the ABT for the variable bound by the abstractor. It is implemented by opening the abstractor to a term with a free variable and then using the spine application form $P{\cdot}S$. The spine supplies the given ABT for the distinguished bound variable and the identity spine for the remainder of the context.

Defining an Indexed Type by Induction: Tries. As an example of an indexed type, we define a generalized trie, which is a dictionary that optimizes its representation based on the type of the key used to index into it, where keys may be products or sums (Hinze, 2000). Tries are represented by indexed family of types trie k d where k is an ABT representing the key and d is the associated data. In pattern-matching notation, the family is defined on normal ABTs as follows:

```
fun trie one d            =  d option
  | trie (prod a1 a2) d   =  trie a1 (trie a2 d)
  | trie (sum a1 a3) d    =  trie a1 d * trie a2 d
```

When the key is unit, the data may or may not be present; when the key is a product, the dictionary is Curried as a map from the first key to a map from the second key to the data; when the key is a sum, the dictionary is represented as a pair of one dictionary for each summand.

This family is defined by induction on the structure of k and non-uniform in d, so we compute a function from types to types with the ABTRec:

```
trie::abt_z[·] -> type -> type = λu.
 ABTRec[_._._.type -> type] (norm (·) z u,
   one    : _._._. λa.a + 1
   prod : _._.v. λa. (fst v) ((snd v) a)
   sum  : _._.v. λa. (fst v) a * (snd v) a
   else =>  void)
```

Because this function inducts over a normal index, it returns the empty type in the (impossible) variable, abstractor, and let cases. In a richer logical framework such as LF, we could define the function norm so that it produces a proof of the normality of the result, and then induct over that evidence instead, which would permit contradicting the impossible cases.

We also define a family of types key[k], classifies the run-time keys mapped to values by the trie:

```
key::abt_z[·] -> type = λu.
 ABTRec[_._._.type] (norm (·) z u,
   one    : _._._. unit
   prod : _._.v. fst v * snd v
   sum  : _._.v. fst v + snd v
   else => void)
```

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):

```
lookup : ∀k::abt_z[·].key k -> ∀d.trie k d -> (d+1)
fun lookup k =
ABTcase[_._.k.key k -> ∀d.trie k d -> (d+1)] (k,
 one : λkey.Λd.λt.t,
 sum : _.p.λkey.Λd.λt.case key of
  { inl key1 => lookup[fst p] key1 [d] (fst t)
  | inr key2 => lookup[snd p] key2 [d] (snd t)},
 ...)
```

The arguments following the ABT are bound inside the ABTRec so that the substitution into the result types of the branches may refine their types. For example, in the first case, unit·() is substituted for k, which causes the type trie k d to reduce to the type d+1; the trie itself is the value we return. In the next case, the substitution propagates the fact that key k is a sum type and that trie k d is a pair; so we may case-analyze the key and look up the subkey in the appropriate dictionary. The well-typedness of this code depends crucially on the computational equalities of the constructor level, which we discuss further in the next section.

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 its types, so that only well-typed terms are representable. In a surface syntax, a small example of such a datatype is as follows:

```
datatype term[a::abt_z[·]] =
  Pair : ∀a,b::abt_z[·]. term[a] -> term[b] ->
         term[prod · (a · it, b · it)]
| Fst  : ∀a,b::abt_z[·].
         term[prod · (a · it, b · it)]
         -> 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 (Crary 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 $(K \to \mathsf{type}) \to (K \to \mathsf{type})$ (though in fact the variable u stands for the $K \to \mathsf{type}$ 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 $(C_1[\lambda\, v.\, \mu_K(u.C_1, v)/u])\ C_2$.

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 term is defined to be the type:

```
λr.μ(term
  (∃a,b::abt_z[·].
     ∃_::Id(prod · (a · it, b · it),r).
      term[a] * term[b])
+ (∃a,b::abt_z[·].∃_::Id(a,r).
    term[prod · (a · it, b · it)]),
r)
```

Using this type, it is straightforward to implement the usual interface of an indexed datatype. For the constructors `Pair` and `Fst`, the summands are simply uncurried versions of the datatype constructor arguments, and the necessary equalities are true by reflexivity. The substitution-based elimination form has the following type:

```
termcase : ∀P::(abt_z[·] -> type).
    (∀a,b::abt_z[·]. term[a] -> term[b]
       -> P(prod · (a · it, b · it)))
 -> (∀a,b::abt_z[·]. term[prod · (a · it, b · it)]
        -> P(a))
 -> ∀u::abt_z[·]. term[u] -> P(u)
```

This operation is implemented as follows:

```
termcase = ΛP.λprC.λfstC.Λu.λt.
case unroll t of
  inl(pack(a,b,pf,(x,y))) => cast P pf (prC a b x y)
| inr(pack(a,b,pf,x)) => cast P pf (fstC a b x)
```

In each branch, we use a nested pattern `pack(a,b,pf,_)` to eliminate the three existentials at once, as an abbreviation for a succession of unpacks. The auxiliary function `cast` uses the exposed proof to pass from the type of the branch to the generic result type; it is implemented as follows:

```
cast : ∀P::(abt_z[·] -> type). ∀_::Id(u,u')
         P(u) -> P(u')
= ΛP.Λpf. Idrec [v._. P(u) -> P(v)] (P, λx.x)
```

This function uses the run-time elimination form for proofs to justify supplying the identity function as the coercion; the details of Idrec are discussed in the next section. In a richer logical framework, a programmer could define non-trivial run-time actions for programmer-defined propositions by using the run-time ABTcase in a similar manner.

5. Formalism

In this section, we give an abridged formal description of our type theory. An online appendix including the full definition of the language will soon be available.[1] We employ a bidirectional type system, inspired by the canonical forms presentations of type theories (Watkins et al., 2002, 2004).

Syntax

In Figure 6, we present the full syntax of our calculus. The syntax of ABTs is as we discussed above except for two technical details. First, in signatures Ω, we use the subsyntax $\hat{L}$ of L, which we define to be those arities that are concrete lists of numerals (i.e., neither computed valences P nor contexts $P \times L$ are permitted). Morally, the signature is defined before any variables in the context, so it cannot mention any computed forms; making this invariant into a syntactic restriction is technically beneficial. Second, we annotate the second projection from a free variable context $P \times L$ with the constructor P being passed over, writing $\pi_2^P\ U$; the role of this annotation is discussed below.

The constructors are stratified into synthesizable constructors P and checked constructors C; the terms are stratified into synthesizable terms R and checked terms E. Note that variables stand for synthesizable constructors/terms. Synthesizable constructors are included into checked constructors, and checked constructors are 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.

We write $N \uparrow$ for and $S \uparrow$ for shifting each U in N by one:

$$\begin{array}{rcl} (\pi_1\ U) \uparrow & = & \pi_1\ (\pi_2\ U) \\ (\mathsf{spn}(U); S) \uparrow & = & \mathsf{spn}(\pi_2\ U); S \\ \text{else} & & \text{defined compositionally} \end{array}$$

$\mathsf{pivot}(L, U)$ decomposes L as $L_1 \times \mathsf{z} \times L_2$ where z corresponds to $\pi_1\ U$:

$$\begin{array}{ll} \mathsf{pivot}(I \times L, \mathsf{it}) = (\cdot, L) & \\ \mathsf{pivot}(I \times L, \pi_2\ U) = (I \times L_1, L_2) & \text{if } \mathsf{pivot}(I, L) = (L_1, L_2) \\ \mathsf{pivot}(P \times L, \pi_2^{P'}\ U) = (P \times L_1, L_2) & \text{if } \mathsf{pivot}(I, L) = (L_1, L_2) \end{array}$$

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

Each step rule has an implicit premise
$\cdot \vdash A[L : \mathsf{ctx}/w][I : \mathsf{vale}/i][P/u] \Downarrow A_{\Downarrow} : \mathsf{type}$
for the I and L in the kind resulting from the normalization of P.

$$\frac{\cdot \vdash P \Downarrow \pi_1\ U : \mathsf{abt}_{\mathsf{z}}(L) \quad \mathsf{pivot}(L, U) = (L_1, L_2)}{\mathsf{ABTcase}[w.u.v.A](P, w.w'.E_1, \_, \_) \mapsto E_1[L_1 : \mathsf{ctx}][L_2 : \mathsf{ctx}] : A_{\Downarrow}}$$

$$\frac{\begin{array}{l}\cdot \vdash P \Downarrow x.\,N : \mathsf{abt}_{\mathsf{s}\ I}(L) \\ E' = E_1[\mathsf{z} \times L : \mathsf{ctx}][I : \mathsf{vale}][([\pi_1\ \mathsf{it}/x](N \uparrow)) : \mathsf{abt}_I(\mathsf{z} \times L)]\end{array}}{\mathsf{ABTcase}[w.i.u.A](P, \_, w.i.u.E_1, \_) \mapsto E' : A_{\Downarrow}}$$

$$\frac{\mathsf{o} : L' \Rightarrow \mathsf{z} \text{ in } \Omega \quad \cdot \vdash P \Downarrow \mathsf{o}{\cdot}S : \mathsf{abt}_{\mathsf{z}}(L) \quad \mathsf{select}(H, \Omega, \mathsf{o}) = w.u.E}{\mathsf{ABTcase}[w.i.u.A](P, \_, \_, H) \mapsto E[L : \mathsf{ctx}][\mathsf{fromspine}(S) : \mathsf{kind}_L(L')] : A_{\Downarrow}}$$

Figure 10. Dynamic Semantics of ABTcase

Judgements

The calculus is defined by the following judgements:

- Valence and ABT context formation are defined by the judgements $\Gamma \vdash I$ valence and $\Gamma \vdash L$ context. The judgements $\Gamma; \Psi; L \vdash N \Leftarrow I$ and $\Gamma; \Psi; L \vdash S \Leftarrow L'$ and $\Gamma; L' \vdash U \Rightarrow L$ 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 $\Gamma \vdash K$ kind, $\Gamma \vdash C \Leftarrow K$, and $\Gamma \vdash P \Rightarrow K$ define kind and constructor formation. The judgement $\Gamma \vdash C \Leftarrow K$ checks a constructor against a known kind, whereas the judgement $\Gamma \vdash P \Rightarrow K$ synthesizes a kind from a constructor. The judgements $\Gamma \vdash K \Downarrow K'$ and $\Gamma \vdash C \Downarrow C' : K$ define kind and constructor normalization. For constructor normalization, Γ, C, and K are inputs, and C' is the output normal form; kind normalization is similar.
- The judgements $\Gamma \vdash E \Leftarrow A$ and $\Gamma \vdash R \Rightarrow A$ define the static semantics of terms. The modes are analogous to those of the constructor judgements.
- The judgements $E \mapsto E'$ and $R \mapsto E : A$ 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 kinding and typing. In our calculus, these equalities are managed by

[1] `http://www.cs.cmu.edu/~drl/`

Abstract Binding Trees:

$$
\begin{array}{llcl}
\textbf{Valence} & I & ::= & \mathsf{z} \mid \mathsf{s}\ I \mid P \\
\textbf{Context} & L & ::= & \cdot \mid I \times L \mid P \times L \\
\textbf{Term} & N & ::= & x \mid x.\,N \mid \mathsf{o}{\cdot}S \mid \pi_1\ U \mid P{\cdot}S \\
\textbf{Spine} & S & ::= & () \mid N; S \mid \mathsf{spn}(U); S \\
\textbf{Free Var.} & U & ::= & \mathsf{it} \mid \pi_2\ U \mid \pi_2^P\ U \\
\textbf{Signature} & \Omega & ::= & \cdot \mid \Omega, \mathsf{o} : \hat{L} \Rightarrow \mathsf{z}
\end{array}
$$

Kinds and Constructors:

$$
\begin{array}{llcl}
\textbf{Kind} & K & ::= & \mathsf{type} \mid \Pi\, u{::}K_1.\,K_2 \mid \Sigma\, u{::}K_1.\,K_2 \mid \mathsf{unit} \\
 & & & \mid \mathsf{Id}_{K_1,K_2}(C_1, C_2) \mid \mathsf{vale} \mid \mathsf{ctx} \mid \mathsf{abt}_I(L) \\
\textbf{Synth. Con.} & P & ::= & u \mid C : K \mid P\ C \mid \mathsf{fst}\ P \mid \mathsf{snd}\ P \mid \mathsf{IdRec}[v.p.K](P, C) \\
 & & & \mid \mathsf{ABTRec}[w.u.v.K](P, w.w'.C_1, w.i.u.v.C_2, B) \\
\textbf{Con. Branch} & B & ::= & \cdot \mid w.u.r.C, B \\
\textbf{Constructor} & A, C & ::= & P \mid A_1 \to A_2 \mid A_1 \times A_2 \mid \mathsf{unit} \mid A_1 + A_2 \mid \mathsf{void} \\
 & & & \mid \forall\, u{::}K.\,A \mid \exists\, u{::}K.\,A \mid \mu_K(u.C_1, C_2) \\
 & & & \mid \lambda\, u.\ C \mid \langle C_1, C_2 \rangle \mid \langle\rangle \mid \mathsf{refl} \mid I \mid L \mid N
\end{array}
$$

Terms:

$$
\begin{array}{llcl}
\textbf{Synth. Term} & R & ::= & x \mid E : A \mid R\ E \mid \mathsf{fst}\ R \mid \mathsf{snd}\ R \mid R[C] \mid \mathsf{unroll}\ R \mid \mathsf{Idrec}[v.p.K](P, E) \mid \mathsf{ABTcase}[w.u.v.A](P, w.w'.E_1, w.i.u.E_2, H) \\
\textbf{Term Branch} & H & ::= & \cdot \mid w.u.E, H \\
\textbf{Term} & E & ::= & f \mid R \mid \lambda\, x.\ E \mid \langle E_1, E_2 \rangle \mid \langle\rangle \mid \mathsf{inl}\ E \mid \mathsf{inr}\ E \mid \mathsf{case}(R, x_1.E_1, x_2.E_2) \mid \mathsf{abort}\ R \\
 & & & \mid \Lambda\, u.\ E \mid \mathsf{pack}(C, E) \mid \mathsf{unpack}(R, u.x.E) \mid \mathsf{roll}\ E \mid \mathsf{fix}\, f.E
\end{array}
$$

Figure 6. Full Syntax

$$\boxed{\Gamma \vdash I\ \mathsf{valence}} \qquad \frac{}{\Gamma \vdash \mathsf{z}\ \mathsf{valence}} \qquad \frac{\Gamma \vdash I\ \mathsf{valence}}{\Gamma \vdash \mathsf{s}\ I\ \mathsf{valence}} \qquad \frac{\Gamma \vdash P :: \mathsf{vale}}{\Gamma \vdash P\ \mathsf{valence}}$$

$$\boxed{\Gamma \vdash L\ \mathsf{context}} \qquad \frac{}{\Gamma \vdash \cdot\ \mathsf{context}} \qquad \frac{\Gamma \vdash L\ \mathsf{context}}{\Gamma \vdash \mathsf{z} \times L\ \mathsf{context}} \qquad \frac{\Gamma \vdash P :: \mathsf{ctx} \quad \Gamma \vdash L\ \mathsf{context}}{\Gamma \vdash P \times L\ \mathsf{context}}$$

$$\boxed{\Gamma; \Psi; L \vdash N \Leftarrow I} \qquad \frac{x\ \mathsf{in}\ \Psi}{\Gamma; \Psi; L \vdash x \Leftarrow \mathsf{z}} \qquad \frac{\Gamma; \Psi, x; L \vdash N \Leftarrow I}{\Gamma; \Psi; L \vdash x.\,N \Leftarrow \mathsf{s}\ I} \qquad \frac{o : L \to \mathsf{z}\ \mathsf{in}\ \Omega \quad \Gamma; \Psi; L \vdash S \Leftarrow L}{\Gamma; \Psi; L \vdash \mathsf{o}{\cdot}S \Leftarrow \mathsf{z}}$$

$$\frac{\Gamma; L \vdash U \Rightarrow I \times L}{\Gamma; \Psi; L \vdash \pi_1\ U \Leftarrow I} \qquad \frac{\Gamma \vdash P \Rightarrow \mathsf{abt}_I(L') \quad \Gamma; \Psi; L \vdash S \Leftarrow L'}{\Gamma; \Psi; L \vdash P{\cdot}S \Leftarrow I}$$

$$\boxed{\Gamma; \Psi; L' \vdash S \Leftarrow L} \qquad \frac{}{\Gamma; \Psi; L \vdash () \Leftarrow \cdot} \qquad \frac{\Gamma; \Psi; L' \vdash N \Leftarrow I \quad \Gamma; \Psi; L' \vdash S \Leftarrow L}{\Gamma; \Psi; L' \vdash N; S \Leftarrow I \times L} \qquad \frac{\Gamma; L' \vdash U \Rightarrow P \times L \quad \Gamma; \Psi; L' \vdash S \Leftarrow L}{\Gamma; \Psi; L' \vdash \mathsf{spn}(U); S \Leftarrow P \times L}$$

$$\boxed{\Gamma; L' \vdash U \Rightarrow L} \qquad \frac{}{\Gamma; L' \vdash \mathsf{it} \Rightarrow L} \qquad \frac{\Gamma; L' \vdash U \Rightarrow I \times L}{\Gamma; L' \vdash \pi_2\ U \Rightarrow L} \qquad \frac{\Gamma; L' \vdash U \Rightarrow P' \times L \quad \Gamma \vdash P \Downarrow P' : \mathsf{ctx}}{\Gamma; L' \vdash \pi_2^P\ U \Rightarrow L}$$

$$\boxed{\Gamma \vdash K\ \mathsf{kind}} \qquad \frac{\begin{array}{l}\Gamma \vdash K_1\ \mathsf{kind} \\ \Gamma \vdash K_1 \Downarrow K_1' \quad \Gamma, u :: K_1' \vdash K_2\ \mathsf{kind}\end{array}}{\Gamma \vdash \Pi\, u{::}K_1.\,K_2\ \mathsf{kind}} \qquad \frac{\begin{array}{lll}\Gamma \vdash K_1\ \mathsf{kind} & \Gamma \vdash K_2\ \mathsf{kind} & \Gamma \vdash C_1 \Leftarrow K_1 \\ \Gamma \vdash K_1 \Downarrow K_1' & \Gamma \vdash K_2 \Downarrow K_2' & \Gamma \vdash C_2 \Leftarrow K_2\end{array}}{\Gamma \vdash \mathsf{Id}_{K_1,K_2}(C_1, C_2)\ \mathsf{kind}} \qquad \frac{\Gamma \vdash I\ \mathsf{valence} \quad \Gamma \vdash L\ \mathsf{context}}{\Gamma \vdash \mathsf{abt}_I(L)\ \mathsf{kind}}$$

$$\boxed{\Gamma \vdash P \Rightarrow K} \qquad \frac{u :: K\ \mathsf{in}\ \Gamma}{\Gamma \vdash u \Rightarrow K} \qquad \frac{\begin{array}{l}\Gamma \vdash K\ \mathsf{kind} \\ \Gamma \vdash K \Downarrow K' \\ \Gamma \vdash C \Leftarrow K'\end{array}}{\Gamma \vdash C : K \Rightarrow K'} \qquad \frac{\begin{array}{ll}\Gamma \vdash P \Rightarrow \mathsf{Id}_{K_1,K_1}(C_1, C_2) & \Gamma \vdash C \Leftarrow K' \\ \Gamma, v :: K_1, p :: \mathsf{Id}_{K_1,K_1}(C_1, v) \vdash K\ \mathsf{kind} & \Gamma \vdash K'[C_2 : K_1/v][P/p] \Downarrow K'' \\ \Gamma \vdash K[C_1 : K_1/v][\mathsf{refl} : \mathsf{Id}_{K_1,K_1}(C_1, C_1)/p] \Downarrow K' & \end{array}}{\Gamma \vdash \mathsf{IdRec}[v.p.K](P, C) \Rightarrow K''}$$

$$\boxed{\Gamma \vdash C \Leftarrow K} \qquad \frac{\Gamma \vdash P \Rightarrow K}{\Gamma \vdash P \Leftarrow K} \qquad \frac{\begin{array}{c}\Gamma \vdash C_1 \Leftarrow K_1 \\ \Gamma \vdash [C_1 : K_1/u]K_2 \Downarrow K_2' \quad \Gamma \vdash C_2 \Leftarrow K_2'\end{array}}{\Gamma \vdash \langle C_1, C_2 \rangle \Leftarrow \Sigma\, u{::}K_1.\,K_2} \qquad \frac{\begin{array}{ll}\Gamma \vdash K\ \mathsf{kind} & \Gamma \vdash K \Downarrow K' \\ \Gamma, u :: K' \to \mathsf{type} \vdash C_1 \Leftarrow K' \to \mathsf{type} & \Gamma \vdash C_2 \Leftarrow K'\end{array}}{\Gamma \vdash \mu_K(u.C_1, C_2) \Leftarrow \mathsf{type}}$$

$$\frac{}{\Gamma \vdash \mathsf{refl} \Leftarrow \mathsf{Id}_{K,K}(C, C)} \qquad \frac{\Gamma \vdash I\ \mathsf{valence}}{\Gamma \vdash I \Leftarrow \mathsf{vale}} \qquad \frac{\Gamma \vdash L\ \mathsf{context}}{\Gamma \vdash L \Leftarrow \mathsf{ctx}} \qquad \frac{\Gamma; \cdot; L \vdash N \Leftarrow I}{\Gamma \vdash N \Leftarrow \mathsf{abt}_I(L)}$$

Figure 7. Selected Kind and Constructor Formation Rules

$$
\begin{array}{rcl}
\mathsf{kind}_{L'}(\cdot) & = & \mathsf{unit} \\
\mathsf{kind}_{L'}(I \times L) & = & \mathsf{abt}_I(L') \times \mathsf{kind}_{L'}(L) \\
 & & \\
\mathsf{tospine}(P, \cdot) & = & () \\
\mathsf{tospine}(P, I \times L) & = & (\mathsf{fst}\ P{\cdot}\mathsf{spn}(\mathsf{it})); \mathsf{tospine}(\mathsf{snd}\ P, L)
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{ih}_{(w, w_0.i_0.u_0.K)}(P, \cdot) & = & \mathsf{unit} \\
\mathsf{ih}_{(w, w_0.i_0.u_0.K)}(P, I \times L) & = & K[w/w_0][I : \mathsf{vale}/i_0][\mathsf{fst}\ P/u_0] \times \\
 & & (\mathsf{ih}_{(w, w_0.i_0.u_0.K)}(\mathsf{snd}\ P, L)) \\
\mathsf{fromspine}(\cdot) & = & \langle\rangle \\
\mathsf{fromspine}(N; L) & = & \langle N, \mathsf{fromspine}(L) \rangle
\end{array}
$$

Figure 8. Auxiliary Operations

$$\frac{\begin{array}{l} \Gamma, w_0 : \mathsf{ctx}, i_0 : \mathsf{vale}, u_0 : \mathsf{abt}_{i_0}(w_0 \times \cdot) \vdash K\ \mathsf{kind} \\ \Gamma \vdash P \Rightarrow \mathsf{abt}_I(L) \\ \Gamma, w : ctx, w' : ctx \vdash K[w \times \mathsf{z} \times w' \times \cdot : \mathsf{ctx}][\mathsf{z} : \mathsf{vale}][\pi_1\ \pi_2\ \mathsf{it} : \mathsf{abt}_{\mathsf{z}}(w \times \mathsf{z} \times w' \times \cdot)] \Downarrow K_1 \\ \Gamma, w : \mathsf{ctx}, w' : \mathsf{ctx} \vdash C_1 \Downarrow K_1 \\ \Gamma, w : \mathsf{ctx}, i : \mathsf{vale}, u : \mathsf{abt}_i(\mathsf{z} \times w \times \cdot) \vdash K[\mathsf{z} \times w \times \cdot : \mathsf{ctx}][i][u] \Downarrow K_2 \\ \Gamma, w : \mathsf{ctx}, i : \mathsf{vale}, u : \mathsf{abt}_i(\mathsf{z} \times w \times \cdot) \vdash K[w][\mathsf{s}\ i : \mathsf{vale}][x.\ u \cdot (x; \mathsf{spn}(w); ()) : \mathsf{abt}_{\mathsf{s}\ i}(w)] \Downarrow K_2' \\ \Gamma, w : \mathsf{ctx}, i : \mathsf{vale}, u : \mathsf{abt}_i(\mathsf{z} \times w \times \cdot), v : K_2 \vdash C_2 \Leftarrow K_2' \\ \Gamma \vdash B : \Omega \rightarrow w_0.i_0.u_0.K \\ \Gamma \vdash K[L : \mathsf{ctx}][I : \mathsf{vale}][P] \Downarrow K' \end{array}}{\Gamma \vdash \mathsf{ABTRec}[w_0.i_0.K](P, w.w'.C_1, w.i.u.v.C_2, B) \Rightarrow K'}$$

$$\boxed{\Gamma \vdash B : \Omega \rightarrow w_0.i_0.u_0.K}$$

$$\frac{}{\Gamma \vdash \cdot : \cdot \rightarrow w_0.i_0.u_0.K} \qquad \frac{\begin{array}{l} \Gamma, w : \mathsf{ctx}, u : \mathsf{kind}_{w \times \cdot}(L) \vdash \mathsf{ih}_{(w, w_0.i_0.u_0.K)}(u, L) \Downarrow K_{ih} \\ \Gamma, w : \mathsf{ctx}, u : \mathsf{kind}_{w \times \cdot}(L) \vdash K[w][\mathsf{z}][\mathsf{o} \cdot \mathsf{tospine}(u, L)] \Downarrow K_{res} \\ \Gamma, w : \mathsf{ctx}, u : \mathsf{kind}_{w \times \cdot}(L), r : K_{ih} \vdash C \Leftarrow K_{res} \\ \Gamma \vdash B : \Omega \rightarrow w_0.i_0.u_0.K \end{array}}{\Gamma \vdash (w.u.r.C, B) : (\mathsf{o} : L \Rightarrow \mathsf{z}, \Omega) \rightarrow w_0.i_0.u_0.K}$$

Figure 9. Kinding of ABTRec

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 Γ 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 calculus'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. The rules for ABT context formation permit variable contexts, but only permit the valence z, as all variables have valence z. The rule for $\pi_2^P\ U$ ensures that the annotation P matches the synthesized (and therefore normal) constructor P' in the classifier using normalization.

Kind formation is as discussed above; observe that the rule for $\mathsf{Id}_{K_1,K_2}(C_1, C_2)$ admits equations between constructors of different kinds. Among the rules for the synthesized constructors, the elimination form for identity may merit some discussion. The construct Idrec is annotated with a result kind pattern $v.p.K$. When P proves $\mathsf{Id}_{K_1,K_1}(C_1, C_2)$, 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 reflexivity 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 Ψ, 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 types, 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 Ω.

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 Ω, and that each branch has the appropriate type. The operation $\mathsf{kind}_{L'}(L)$ maps the arity of the operator into a product kind of ABTs of the correct valence; it is used to compute the kind of the exposed subterms. The operation $\mathsf{ih}_{(w,\_.K)}(P, L)$ computes (a non-normalized form of) the kind of the inductive hypothesis. Finally, the operation $\mathsf{tospine}(P, L)$ 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 $w \times \mathsf{z} \times w' \times \cdot$ 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 Idrec and ABTcase, 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 $\beta\eta$-normal forms for the kinds $\Pi\ u{::}K_1 .\ K_2$, $\Sigma\ u{::}K_1 .\ K_2$, and unit, and β-normal forms for $\mathsf{Id}_{K_1,K_2}(C_1, C_2)$ and $\mathsf{abt}_I(L)$.

Additionally, the algorithm reduces computed ABTs, valences and contexts (equating, for example, $\mathsf{s}\,(\mathsf{s}\ i : \mathsf{vale})$ with $\mathsf{s}\,(\mathsf{s}\ i)$). The algorithm maintains the product structure L representing the ABT context in our chosen list-like representation. For example, the context $(\mathsf{ctx}(\mathsf{z} \times \mathsf{z}) : \mathsf{ctx}) \times \mathsf{z}$ is normalized to the context $\mathsf{z} \times (\mathsf{z} \times (\mathsf{z} \times \cdot))$. Because reassociating the products changes the meaning of the projections, the normalization algorithm must rewrite them; the annotation on $\pi_2^P\ U$ 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 $R \mapsto E : A$, a synthesizing term steps to a checked term E and produces a type annotation A, which is used to annotate E if it needs to be regarded as a synthesizing term (e.g., in the congruence rule for the first position of application). We show the β rules for $\mathsf{ABTcase}$ 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* $\cdot \vdash A \Leftarrow \mathsf{type}$ *and* $\cdot \vdash A \Downarrow A : \mathsf{type}$.

> ***Preservation*** *If* $E \mapsto E'$ *and* $\cdot \vdash E \Leftarrow A$ *then* $\cdot \vdash E' \Leftarrow A$.
> ***Progress*** *If* $\cdot \vdash E \Leftarrow A$ *then* $E \mapsto E'$ *or* E value.

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, 2004; Nanevski et al., 2006; 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 $\mathsf{P}\cdot S$, 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 abstractor 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 α-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 describe 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 $\mathsf{abt}_I(L)$, the recursor 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

L. Augustsson. Cayenne - a language with dependent types. In *International Conference on Functional Programming*, 1998.

C. Chen and H. Xi. Combining programming with theorem proving. In *International Conference on Functional Programming*, 2005.

J. Cheney and R. Hinze. Phantom types. Technical Report CUCIS TR20003-1901, Cornell University, 2003.

B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *Programming Language Design and Implementation*, 2005.

J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *European Symposium on Programming*, 2007.

R. L. Constable and N. P. Mendler. Recursive definitions in type theory. In *Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 61–78. Springer-Verlag, 1985.

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

T. Coquand and G. P. Huet. The calculus of constructions. *Information and Computation*, 76(2/3), 1988.

K. Crary and S. Weirich. Flexible type analysis. In *International Conference on Functional Programming*, 1999.

J. Dunfield and F. Pfenning. Tridirectional typechecking. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2004.

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

C. Flanagan. Hybrid type checking. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–256, 2006.

S. Fogarty, E. Pasalic, J. Siek, and W. Taha. Concoqtion: indexed types now! In *ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 112–121, New York, NY, USA, 2007. ACM Press. ISBN 978-1-59593-620-2.

R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Symposium on Principles of Programming Languages*, 1995.

R. Harper and F. Pfenning. On equivalence and canonical forms in the LF type theory. *ACM Transactions on Computational Logic*, 6:61–101, 2005.

R. Harper and C. Stone. A type-theoretic interpretation of Standard ML. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press, 2000.

R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *Symposium on Principles of Programming Languages*, 1990.

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

R. Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10:327–351, 2000.

K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *European Conference on Object-Oriented Programming*, pages 491–516, Oslo, Norway, 2004.

D. R. Licata and R. Harper. A formulation of Dependent ML with explicit equality proofs. Technical Report CMU-CS-05-178, Department of Computer Science, Carnegie Mellon University, 2005.

Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*, volume 11 of *International Series of Monographs on Computer Science*. Oxford University Press, 1994.

C. McBride. *Dependently Typed Functional Programs and Their Proofs*. PhD thesis, University of Edinburgh, 2000.

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

A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare Type Theory. In *icfp*, pages 62–73, Portland, Oregon, 2006.

A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 2007. To appear.

E. Pasalic. *The Role of Type Equality in Meta-Programming*. PhD thesis, Oregon Health & Science University, OGI School of Science & Engineering, 2004.

L. Petersen. *Certifying Compilation for Standard ML in a Type Analysis Framework*. PhD thesis, Carnegie Mellon University, 2005.

S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ACM SIGPLAN International Conference on Functional Programming*, 2006.

B. Pientka. Functional programming with higher-order abstract syntax and explicit substitutions. In *Programming Languages meets Program Verification*, 2006.

A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

A. M. Pitts. Alpha-structural recursion and induction. *Journal of the Association for Computing Machinery*, 53:459–506, 2006.

A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, Heidelberg, 2000.

F. Pottier. Static name control for FreshML. In *IEEE Symposium on Logic in Computer Science*, 2007.

S. Sarkar. A cost-effective foundational certified code system. Thesis Proposal, Carenegie Mellon University, 2005.

C. Schürmann and F. Pfenning. A coverage checking algorithm for LF. In *International Conference on Theorem Proving in Higher-Order Logics*. Springer-Verlag, 2003.

C. Schürmann, J. Despeyroux, and F. Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, 266:1–57, 2001.

C. Schürmann, A. Poswolsky, and J. Sarnat. The ∇-calculus: Functional programming with higher-order encodings. In *International Conference on Typed Lambda Calculi and Applications*. Springer-Verlag, 2005.

Z. Shao, V. Trifonov, B. Saha, and N. Papaspyrou. A type system for certified binaries. *ACM Transactions on Programming Languages and Systems*, 27(1):1–45, 2005.

T. Sheard. Languages of the future. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

M. Sulzmann, M. Chakravarty, and S. Peyton Jones. System f with type equality coercions. In *ACM SIGPLAN-SIGACT Symposium on Types in Language Design and Implementation*, 2007.

K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework: the propositional fragment. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 355–377. Springer-Verlag, 2004.

S. Weirich. Higher-order intensional type analysis. In D. L. Mt́ayer, editor, *European Symposium on Programming*, pages 98–114, 2002.

E. Westbrook, A. Stump, and I. Wehrman. A language-based approach to functionally correct imperative programming. In *International Conference on Functional Programming*, 2005.

H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Conference on Programming Language Design and Implementation*, 1998.

H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2003.

C. Zenger. *Indizierte Typen*. PhD thesis, Universit at Karlsruhe, 1998.