A Type System for Coordinated Data Structures

Michael F. Ringenburg* Dan Grossman
Department of Computer Science & Engineering
University of Washington, Seattle, WA 98195
{miker,djg}@cs.washington.edu

Abstract

Low-level type systems aim to offer great flexibility in the choice of a program's data representations. However, conventional wisdom suggests that low-level, polymorphic type systems cannot naturally support data representations that involve the sharing of existentially quantified types between corresponding nodes of separate recursive data structures (herein referred to as *coordinated data structures*). In this paper, we do just that: We show how a standard, low-level, polymorphic type system can be modified to support coordinated data structures by enriching recursive types and adding type trees. We prove the soundness of our modification and illustrate its power with examples, including "tagless" lists and red-black trees where the values and colors are stored in separate trees that are guaranteed to have the same shape.

1 Introduction

This paper extends conventional polymorphic typed λ-calculi in a natural way such that the type systems can express invariants between coordinated recursive data structures. Examples of such invariants include two trees with identical shape, or a list of function pointers and a separate list of corresponding environment records (where the i-th environment record corresponds to the i-th function). The rest of this section motivates why such invariants are important, explores why the scope of type variables makes the problem appear daunting, and discusses why our solution is both simple and powerful.

*Supported in part by an Achievement Rewards for College Scientists (ARCS) fellowship sponsored by the Washington Research Foundation (WRF).

1.1 Low-Level Type Systems

Recent years have witnessed substantial work on powerful type systems for safe, low-level languages. Standard motivation for such systems includes compiler debugging (generated code that does not type check implies a compiler error), proof-carrying code (the type system encodes a safety property that the type-checker verifies), automated optimization (an optimizer can exploit the type information), and manual optimization (humans can use idioms unavailable in higher-level languages without sacrificing safety). An essential difference between high- and low-level languages is that the latter have *explicit data representations*; implementations are not at liberty to add fields or levels of indirection. Compilers for high-level languages *encode* constructs (e.g., function closures) explicitly (e.g., as a pair of a code pointer and an environment record of free-variable values).

For many reasons, including the belief that data-representation decisions affect performance, low-level type systems aim to allow great flexibility in making these decisions. But as usual, the demands of efficient type-checking limit the possible encodings. Type systems striking an attractive balance between data-representation flexibility and straightforward checking have typically been based on typed λ-calculi with powerful constructors for sum types, recursive types, universal types, existential types, and (higher-order) type constructors. In such calculi, we can encode data structures such as lists of closures without the type system mandating the representation of lists or closures.

1.2 Type-Variable Scope

Unfortunately, low-level typed λ-calculi have suffered from an embarrassing restriction resulting from the scope of type variables. For example, consider encoding functions that have type $\mathsf{int} \rightarrow \mathsf{int}$ in

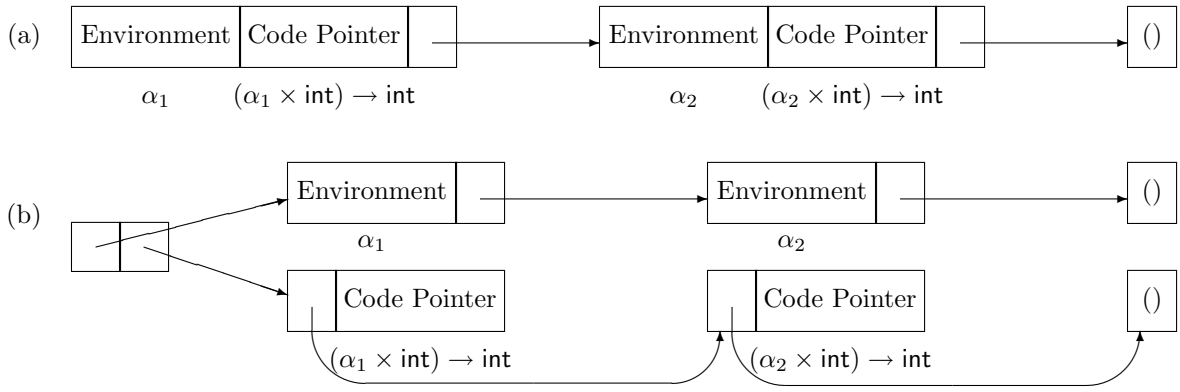

Figure 1: A list of function closures encoded as (a) a single list, and (b) a coordinated pair of lists (using two different list representations).

a typed functional language such as Haskell or ML. A simple encoding is $\exists\alpha.\ \alpha \times ((\alpha \times \mathsf{int}) \to \mathsf{int})$. We abstract over the type of a data structure storing the values of the function's free variables and use a pair holding this structure and a closed function taking the structure and an `int` [14]. (Whether pair types add a level of indirection is important in low-level languages, but not for this paper.) The existential quantifier is crucial for ensuring all functions of type $\mathsf{int} \to \mathsf{int}$ in the source language have the same type after compilation (even if their environments have different types), which allows functions to be first-class. For example, a list of such functions could have type $\mu\beta.1 + ((\exists\alpha.\ \alpha \times ((\alpha \times \mathsf{int}) \to \mathsf{int})) \times \beta)$ (or another list encoding), where $\alpha + \beta$ represents a sum type with variants α and β, and 1 represents the unit type. Figure 1a displays this encoding.

But suppose we want *two coordinated lists* (as in Figure 1b) in which one list holds environment records (the αs) and the other holds code pointers (the $\to$s), with the i^{th} element of one list being the record for the i^{th} element of the other. This choice may seem silly for a functional-language compiler, but there are many reasons why we may wish to "distribute" an existentially bound tuple across "coordinated" data structures (such as lists):

- Legacy code: We may be conceptually adding a field to existing types but be unable to recompile parts of our system. We can do this by leaving arrays of such records unchanged and using a "parallel array" to hold the new field.
- Cache behavior: If some fields are rarely accessed, we may place them in a separate data structure to reduce working-set size.
- Data packing: Collections of records with fields of different sizes can be stored more efficiently by segregating the fields rather than the records. For example, a pair of a one-bit and a 32-bit value often consumes 64-bits of space due to alignment restrictions.

The most important reasons are the ones we have not thought of: The purpose of low-level type systems is to allow "natural" data representations without planning for them in advance. In low-level code, there is nothing unnatural about coordinated data structures.

At first glance, polymorphic type systems seem ill-equipped to allow this flexibility: To abstract a type, we must choose a scope for the type variable. For coordinated lists, we need a scope encompassing the lists. But the lists have unbounded size, thus the scope may be unbounded.

This limitation is fairly well-known, but to our knowledge it has remained because it was unclear how to remove it in a way that "fit" with what are already sophisticated systems. It turns out Crary and Weirich's formal language LX [6] can actually encode coordinated data structures like the ones described here (see Section 9), but it is more complex than our extension and was not considered for this purpose.

1.3 A Surprisingly Easy Extension

This paper describes a simple type-theoretic way to allow coordinated data structures. It enjoys the following strengths:

- Modest type-language extensions: We use a simple form of parameterized recursive types. The other type constructors are unchanged.

We also add kinds for infinite collections of types, which circumvent the type-variable scope problem. Low-level type systems already include kinds.

- Modest term-language extensions: We add only one new term (called "peel"), which is essentially a coercion on coordinated data structures that rewrites their types in a particular way. The typing rules for other constructs are unchanged (modulo the form of recursive types). The peel coercion has a straightforward type-erasure interpretation as function application (much like an existential unpack).
- Expressiveness: The extension is general; it allows coordinated recursive types to abstract over an infinite number of types. The recursive types need not be the same (e.g., one could be a tree and another a list). The extension is synergistic with type variables of unconventional kinds, especially singleton integers.

In short, we have a straightforward localized way to increase the data-representation flexibility of low-level languages. In this paper, we focus on the key idea by using it in a typed λ-calculus, establishing the necessary metatheory, and demonstrating its power via examples. We are confident the idea can carry over to its intended use in safe low-level languages. Additional work could make the technique suitable for human-generated code.

1.4 Outline

The rest of this paper:

- Explains how we use type lists, an enriched form of recursive type, and a new "peel" coercion to circumvent the type-variable scoping issues (Section 2).
- Builds progressively more complex languages, starting with a language for coordinated lists (Section 3), extending it with singleton integers (Section 6), and finally supporting more general recursive coordinated types (Section 7). We illustrate each language with an extended example.
- Establishes type safety and type erasure for the simple coordinated list language (Section 4).
- Considers why it is difficult to encode conventional recursive types with our modified ones, and describes some modest language extensions that make such an encoding possible (Section 5).
- Discusses our prototype implementation (Section 8), related work (Section 9), and future work (Section 10).

2 The Trick

The essence of coordinated data structures is that they assume a potentially unbounded number of type equalities. For example, two lists may assume their i^{th} elements have some connection (such as if one has type β then the other has type $\beta \to \mathsf{int}$ for some β). Conventional type systems can describe potentially unbounded data structures with a recursive type, $\mu\alpha.\tau$, that has finite size. We change conventional recursive types to a simple form of parameterized recursive type:

$$\mu(\sigma \leftarrow \beta)\alpha.\tau$$

where σ is an *infinite-list of types* and, later, an infinite-tree of types, and β (and α) are bound in τ. Intuitively, on the i^{th} unrolling of a recursive type, we substitute the i^{th} element of σ for β. The typing rule for an unroll coercion is therefore the following, where $\tau[\tau'/\alpha]$ is capture-avoiding substitution of τ' for α in τ:

$$\textsc{unroll}\ \frac{\Delta;\Gamma \vdash_\tau e : \mu(\tau'::\sigma' \leftarrow \beta)\alpha.\tau}{\Delta;\Gamma \vdash_\tau \mathsf{unroll}\ e : \tau[\tau'/\beta][\mu(\sigma' \leftarrow \beta)\alpha.\tau/\alpha]}$$

That is, if σ is some $\tau' :: \sigma'$ (a list beginning with τ'), then the unroll coercion substitutes τ' for β and $\mu(\sigma' \leftarrow \beta)\alpha.\tau$ for α, so the next unroll will use the next element of σ (i.e., the first element of σ'). The roll coercion is, as usual, the inverse of unroll:[1]

$$\textsc{roll}\ \frac{\Delta;\Gamma \vdash_\tau e : \tau[\tau'/\beta][\mu(\sigma' \leftarrow \beta)\alpha.\tau/\alpha]}{\Delta;\Gamma \vdash_\tau \mathsf{roll}\ e\ \mathsf{as}\ \mu(\tau'::\sigma' \leftarrow \beta)\alpha.\tau : \mu(\tau'::\sigma' \leftarrow \beta)\alpha.\tau}$$

Both rules reduce to the conventional rules for recursive types provided β does not occur free in τ and we ignore the type lists.

To express that two (or more) data structures are coordinated, we just use the same σ. The example from Figure 1b separating closure-environments and code pointers into coordinated lists is

$$\begin{array}{ll} & (\mu(\sigma \leftarrow \beta)\alpha.1 + \beta \times \alpha) \\ \times & (\mu(\sigma \leftarrow \beta)\alpha.1 + \alpha \times ((\beta \times \mathsf{int}) \to \mathsf{int})) \end{array}$$

But adding just σ and β accomplishes nothing: The type of an unbounded data structure would include

[1] The result type must be well-formed; the rule in Section 3 includes the necessary technical condition.

VARIABLES	x	$\in$	Var
TYPE VARIABLES	α, β	$\in$	Tyvar
KINDS	κ	::=	T | L
TYPES	σ, τ	::=	$1 \mid \alpha \mid \tau \times \tau \mid \tau + \tau \mid \tau \to \tau \mid \forall\alpha{:}\kappa.\tau \mid \exists\alpha{:}\kappa.\tau \mid \mu(\sigma \leftarrow \beta)\alpha.\tau \mid \tau^* \mid \tau{::}\sigma$
EXPRESSIONS	e	::=	$() \mid x \mid (e, e) \mid \pi_i\ e \mid \mathsf{in}_i\ e \mid \mathsf{case}\ e\ \mathsf{of}\ x.e\ x.e \mid \lambda x{:}\tau.\ e \mid e\ e \mid \mathsf{fix}\ e$
		|	$\Lambda\alpha{:}\kappa.\ e \mid e\ [\tau] \mid \mathsf{pack}\ \tau, e\ \mathsf{as}\ \tau \mid \mathsf{unpack}\ e\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ e$
		|	$\mathsf{roll}\ e\ \mathsf{as}\ \tau \mid \mathsf{unroll}\ e \mid \mathsf{peel}\ e\ \mathsf{as}\ \alpha, \alpha, x\ \mathsf{in}\ e$
VALUES	v	::=	$() \mid (v, v) \mid \mathsf{in}_i\ v \mid \lambda x{:}\tau.\ e \mid \Lambda\alpha{:}\kappa.\ e \mid \mathsf{pack}\ \tau, v\ \mathsf{as}\ \tau \mid \mathsf{roll}\ v\ \mathsf{as}\ \tau$
TYPE CONTEXTS	Γ	::=	$\cdot \mid \Gamma, x{:}\tau$
KIND CONTEXTS	Δ	::=	$\cdot \mid \Delta, \alpha{:}\kappa$

Figure 2: The syntax for a language supporting coordinated lists.

a σ of unbounded size. Fortunately, many uses of coordinated data structures need not know the elements of σ, only that the coordinated data structures use the *same* σ. Hence, it suffices to abstract over lists of types, using ordinary existential quantification. For example:

$$\exists\beta'{:}\mathrm{L}.(\ \ (\mu(\beta' \leftarrow \beta)\alpha.1 + \beta \times \alpha) \\ \times(\mu(\beta' \leftarrow \beta)\alpha.1 + \alpha \times ((\beta \times \mathsf{int}) \to \mathsf{int})))$$

where L is a *kind* annotation indicating that β' represents a list of types. Adding this kind to a type system that already has kinds requires no changes to the typing rules for quantified types.

However, our typing rule for unroll does not apply to types of the form $\mu(\beta' \leftarrow \beta)\alpha.\tau$, so there is not yet a way to do anything useful with the pair obtained from unpacking a value with the existential type above. We need a way to replace the β' with some $\alpha_{hd}{::}\alpha_{tl}$ (where α_{hd} has kind T, the kind of conventional types, and α_{tl} has kind L). Most crucially, given multiple types that are coordinated in that they use the same β', we need to replace the β' with the *same* α_{hd} and α_{tl} lest we forget the very invariant we aim to track. We introduce a "peel" coercion (as in peeling α_{hd} off an unknown list) for this purpose:

$$\frac{\begin{array}{l}\Delta; \Gamma \vdash_\tau e_1 : (\mu(\sigma \leftarrow \beta)\alpha.\tau_1) \times (\mu(\sigma \leftarrow \beta)\alpha.\tau_2) \\ \Delta, \alpha_{hd}{:}\mathrm{T}, \alpha_{tl}{:}\mathrm{L}; \Gamma, x{:}(\mu(\alpha_{hd}{::}\alpha_{tl} \leftarrow \beta)\alpha.\tau_1) \times \\ \qquad\qquad (\mu(\alpha_{hd}{::}\alpha_{tl} \leftarrow \beta)\alpha.\tau_2) \\ \qquad\qquad \vdash_\tau e_2 : \tau\end{array}}{\Delta; \Gamma \vdash_\tau \mathsf{peel}\ e_1\ \mathsf{as}\ \alpha_{hd}, \alpha_{tl}, x\ \mathsf{in}\ e_2 : \tau}\ \text{PEEL}$$

This rule allows two coordinated data structures; in practice peel should allow an n-tuple. As Section 3 shows, the coercion never fails at run-time.

In summary, we have introduced type-lists, a kind for abstracting over them, an enrichment of recursive types, and a special peel coercion.

3 A Language For Coordinated Lists

In this section, we present a simple language containing our extensions. Sections 3.1, 3.2, and 3.3 present, respectively, the syntax, semantics, and typing rules. Section 3.4 illustrates the language with an example involving function closures.

We emphasize that this simple language is powerful enough to encode only coordinated data structures where each node has at most one recursive child[2] (e.g., lists). In Section 7, we generalize the language to support coordinated data structures with multiple children (e.g., trees).

3.1 Syntax

Figures 2 defines the syntax for our simple language. Expressions (e) can be: () for unit, x for variables, (e, e) for pairs, $\pi_i\ e$ for projection, $\mathsf{in}_i\ e$ for injection into a sum type, $\mathsf{case}\ e\ \mathsf{of}\ x.e\ x.e$ for branching based on sum types, $\lambda x{:}\tau.\ e$ for functions, $e\ e$ for function application, or $\mathsf{fix}\ e$ for recursion. We also have four cases for introducing and eliminating universally and existentially quantified types. Finally, we have roll and unroll coercions for recursive types, and the previously mentioned peel coercion.

Types (τ or σ) also contain the standard forms, including 1 for unit, $\tau \times \tau$ for pair types, $\tau + \tau$ for sum types, $\forall\alpha{:}\kappa.\tau$ for universally quantified types, and $\exists\alpha{:}\kappa.\tau$ for existentially quantified types. We also have the enriched recursive types described in Section 2. The last two cases, τ^* and $\tau{::}\sigma$, indicate type lists. The type τ^* represents an infinite list of τs, and $\tau{::}\sigma$ represents the list created by adding τ to the head of the type list σ. The kind T represents

[2]Technically, multiple recursive children can be supported, but only if the coordinated types are identical in every child. See Section 7.

$$
\begin{array}{rcl}
E & ::= & [\cdot] \mid (E, e) \mid (v, E) \mid \pi_i\ E \mid \mathsf{in}_i\ E \mid \mathsf{case}\ E\ \mathsf{of}\ x.e\ x.e \mid E\ e \mid v\ E \mid \mathsf{fix}\ E \\
& \mid & E\ [\tau] \mid \mathsf{pack}\ \tau, E\ \mathsf{as}\ \tau \mid \mathsf{unpack}\ E\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ e \\
& \mid & \mathsf{roll}\ E\ \mathsf{as}\ \tau \mid \mathsf{unroll}\ E \mid \mathsf{peel}\ E\ \mathsf{as}\ \alpha, \beta, x\ \mathsf{in}\ e
\end{array}
$$

$$
\begin{array}{rcll}
\pi_i\ (v_1, v_2) & \xrightarrow{\mathrm{r}} & v_i & \text{where } i \in \{1,2\} \\
\mathsf{case}\ \mathsf{in}_i\ v\ \mathsf{of}\ x.e_1\ x.e_2 & \xrightarrow{\mathrm{r}} & e_i[v/x] & \text{where } i \in \{1,2\} \\
(\lambda x{:}\tau.\ e)\ v & \xrightarrow{\mathrm{r}} & e[v/x] & \\
\mathsf{fix}\ \lambda x{:}\tau.\ e & \xrightarrow{\mathrm{r}} & e[(\mathsf{fix}\ \lambda x{:}\tau.\ e)/x] & \\
(\Lambda\alpha{:}\kappa.\ e)\ [\tau] & \xrightarrow{\mathrm{r}} & e[\tau/\alpha] & \\
\mathsf{unpack}\ (\mathsf{pack}\ \tau_1, v\ \mathsf{as}\ \exists\alpha{:}\kappa.\tau_2)\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ e & \xrightarrow{\mathrm{r}} & e[\tau_1/\alpha][v/x] & \\
\mathsf{unroll}\ \mathsf{roll}\ v\ \mathsf{as}\ \tau & \xrightarrow{\mathrm{r}} & v &
\end{array}
$$

$$
\begin{array}{l}
\mathsf{peel}\ (\mathsf{roll}\ v_1\ \mathsf{as}\ \mu(\sigma \leftarrow \beta)\alpha.\tau_1, \mathsf{roll}\ v_2\ \mathsf{as}\ \mu(\sigma \leftarrow \beta)\alpha.\tau_2)\ \mathsf{as}\ \alpha_{hd}, \alpha_{tl}, x\ \mathsf{in}\ e \\
\xrightarrow{\mathrm{r}} \quad e[\tau'/\alpha_{hd}][\sigma'/\alpha_{tl}][(\mathsf{roll}\ v_1\ \mathsf{as}\ \mu(\tau'{::}\sigma' \leftarrow \beta)\alpha.\tau_1, \mathsf{roll}\ v_2\ \mathsf{as}\ \mu(\tau'{::}\sigma' \leftarrow \beta)\alpha.\tau_2)/x] \\
\text{where } \mathrm{peel}(\sigma) = \tau'{::}\sigma'
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{peel}(\tau{::}\sigma) & = & \tau{::}\sigma \\
\mathrm{peel}(\tau^*) & = & \tau{::}\tau^*
\end{array}
\qquad\qquad
\frac{e \xrightarrow{\mathrm{r}} e'}{E[e] \rightarrow E[e']}
$$

Figure 3: The operational semantics for our coordinated-list language.

conventional types, and the kind L represents lists of conventional types.

3.2 Semantics

Figure 3 presents the operational semantics for our coordinated list language. The term-substitution notation $e_1[e_2/x]$ signifies capture-avoiding substitution of e_2 for x in e_1. Similarly, we use $\tau_1[\tau_2/\alpha]$ for type substitution. The semantics uses evaluation contexts (E) to specify order of evaluation (see, for instance, [25]). With the exception of the peel coercion, the rules are fairly standard.

The peel coercion takes a pair of recursively typed values with identical type lists and applies the partial metafunction peel(σ) (defined for all closed types of kind L) to split the type list into a head and a tail. The reduction produces the expression e, modified by substituting the list head for α_{hd}, the list tail for α_{tl}, and the input pair (with peeled lists) for x. A peel coercion never fails at run-time, because all types of kind L represent infinite-list types. If peel could fail, the type-erasure theorem in Section 4 would not hold. As the example in Section 3.4 shows, we can use 1^* (or another *infinite*-list type) for *finite* terms like the empty-list.

3.3 Typing Rules

Typing judgments have the form $\Delta; \Gamma \vdash_{\mathrm{t}} e : \tau$, where Δ is the kind environment and Γ is the type environment. We implicitly assume Δ and Γ have no repeated elements. To avoid naming conflicts, we can systematically rename binding occurrences. The rules ensure every variable in Γ has kind T under Δ. Figure 4 presents the typing rules for our extended language. We use the notation

$$
\frac{\mathcal{P}_1}{\mathcal{P}_2}\ \text{ as shorthand for }\ \frac{\mathcal{P}_1}{\mathcal{P}_2}\ \text{ and }\ \frac{\mathcal{P}_1}{\mathcal{P}_3}\ .
$$

The KSTAR and KCONS rules imply that list types (types of kind L) can be constructed by either applying the * operator to a conventional type (a type of kind T), or by applying the :: operator to a conventional type followed by a list type. The KMU rule states that the type list of a recursive type must have kind L, and that if we assume the bound type variables (α and β) have kind T, then the body (τ) must also have type T. It is correct to assume that β has type T, because the KSTAR and KCONS rules guarantee that the list σ from which β is instantiated will be composed of types of kind T.

The PEEL rule states that the peeled expression (e_1) must be a pair of recursively typed expressions with identical type lists. The type lists are replaced with $\alpha_{hd}{::}\alpha_{tl}$, and the resulting type is assumed for x in e_2 (similar to how x is assigned type τ' in the expression e_2 for unpack coercions). The ROLL and UNROLL rules are explained in Section 2. The rest of the rules have their standard forms. Thus our extension requires only modest changes to the type system. The system also remains syntax-directed,

$\boxed{\Delta \vdash_{\mathrm{k}} \tau : \kappa}$

$$\text{KBASE}\ \frac{}{\Delta \vdash_{\mathrm{k}} 1 : \mathrm{T}} \qquad \Delta \vdash_{\mathrm{k}} \alpha : \Delta(\alpha)$$

$$\text{KPAIR}\ \frac{\Delta \vdash_{\mathrm{k}} \tau_1 : \mathrm{T} \qquad \Delta \vdash_{\mathrm{k}} \tau_2 : \mathrm{T}}{\Delta \vdash_{\mathrm{k}} \tau_1 \times \tau_2 : \mathrm{T} \qquad \Delta \vdash_{\mathrm{k}} \tau_1 + \tau_2 : \mathrm{T} \qquad \Delta \vdash_{\mathrm{k}} \tau_1 \to \tau_2 : \mathrm{T}}$$

$$\text{KQUANT}\ \frac{\Delta, \alpha{:}\kappa \vdash_{\mathrm{k}} \tau : \mathrm{T}}{\Delta \vdash_{\mathrm{k}} \forall \alpha{:}\kappa.\tau : \mathrm{T} \qquad \Delta \vdash_{\mathrm{k}} \exists \alpha{:}\kappa.\tau : \mathrm{T}}$$

$$\text{KMU}\ \frac{\Delta \vdash_{\mathrm{k}} \sigma : \mathrm{L} \qquad \Delta, \alpha{:}\mathrm{T}, \beta{:}\mathrm{T} \vdash_{\mathrm{k}} \tau : \mathrm{T}}{\Delta \vdash_{\mathrm{k}} \mu(\sigma \leftarrow \beta)\alpha.\tau : \mathrm{T}} \qquad \text{KSTAR}\ \frac{\Delta \vdash_{\mathrm{k}} \tau : \mathrm{T}}{\Delta \vdash_{\mathrm{k}} \tau^* : \mathrm{L}} \qquad \text{KCONS}\ \frac{\Delta \vdash_{\mathrm{k}} \tau : \mathrm{T} \qquad \Delta \vdash_{\mathrm{k}} \sigma : \mathrm{L}}{\Delta \vdash_{\mathrm{k}} \tau{::}\sigma : \mathrm{L}}$$

$\boxed{\Delta; \Gamma \vdash_{\mathrm{t}} e : \tau}$

$$\text{BASE}\ \frac{}{\Delta; \Gamma \vdash_{\mathrm{t}} () : 1} \qquad \Delta; \Gamma \vdash_{\mathrm{t}} x : \Gamma(x)$$

$$\text{PAIR}\ \frac{\Delta; \Gamma \vdash_{\mathrm{t}} e_1 : \tau_1 \qquad \Delta; \Gamma \vdash_{\mathrm{t}} e_2 : \tau_2}{\Delta; \Gamma \vdash_{\mathrm{t}} (e_1, e_2) : \tau_1 \times \tau_2} \qquad \text{PROJ}\ \frac{\Delta; \Gamma \vdash_{\mathrm{t}} e : \tau_1 \times \tau_2}{\Delta; \Gamma \vdash_{\mathrm{t}} \pi_1\ e : \tau_1 \qquad \Delta; \Gamma \vdash_{\mathrm{t}} \pi_2\ e : \tau_2}$$

$$\text{INJECT}\ \frac{\Delta; \Gamma \vdash_{\mathrm{t}} e : \tau \qquad \Delta \vdash_{\mathrm{k}} \tau' : \mathrm{T}}{\Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{in}_1\ e : \tau + \tau' \qquad \Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{in}_2\ e : \tau' + \tau} \qquad \text{CASE}\ \frac{\Delta; \Gamma \vdash_{\mathrm{t}} e : \tau_1 + \tau_2 \qquad \Delta; \Gamma, x{:}\tau_1 \vdash_{\mathrm{t}} e_1 : \tau \qquad \Delta; \Gamma, x{:}\tau_2 \vdash_{\mathrm{t}} e_2 : \tau}{\Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{case}\ e\ \mathsf{of}\ x.e_1\ x.e_2 : \tau}$$

$$\text{FUN}\ \frac{\Delta; \Gamma, x{:}\tau \vdash_{\mathrm{t}} e : \tau' \qquad \Delta \vdash_{\mathrm{k}} \tau : \mathrm{T}}{\Delta; \Gamma \vdash_{\mathrm{t}} \lambda x{:}\tau.\ e : \tau \to \tau'} \qquad \text{APP}\ \frac{\Delta; \Gamma \vdash_{\mathrm{t}} e_1 : \tau' \to \tau \qquad \Delta; \Gamma \vdash_{\mathrm{t}} e_2 : \tau'}{\Delta; \Gamma \vdash_{\mathrm{t}} e_1\ e_2 : \tau} \qquad \text{FIX}\ \frac{\Delta; \Gamma \vdash_{\mathrm{t}} e : \tau \to \tau}{\Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{fix}\ e : \tau}$$

$$\text{TFUN}\ \frac{\Delta, \alpha{:}\kappa; \Gamma \vdash_{\mathrm{t}} e : \tau}{\Delta; \Gamma \vdash_{\mathrm{t}} \Lambda\alpha{:}\kappa.\ e : \forall \alpha{:}\kappa.\tau} \qquad \text{TAPP}\ \frac{\Delta; \Gamma \vdash_{\mathrm{t}} e_1 : \forall \alpha{:}\kappa.\tau' \qquad \Delta \vdash_{\mathrm{k}} \tau : \kappa}{\Delta; \Gamma \vdash_{\mathrm{t}} e\ [\tau] : \tau'[\tau/\alpha]}$$

$$\text{PACK}\ \frac{\Delta; \Gamma \vdash_{\mathrm{t}} e : \tau'[\tau/\alpha] \qquad \Delta \vdash_{\mathrm{k}} \tau : \kappa \qquad \Delta \vdash_{\mathrm{k}} \exists \alpha{:}\kappa.\tau' : \mathrm{T}}{\Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{pack}\ \tau, e\ \mathsf{as}\ \exists \alpha{:}\kappa.\tau' : \exists \alpha{:}\kappa.\tau'}$$

$$\text{UNPACK}\ \frac{\Delta; \Gamma \vdash_{\mathrm{t}} e_1 : \exists \alpha{:}\kappa.\tau' \qquad \Delta, \alpha{:}\kappa; \Gamma, x{:}\tau' \vdash_{\mathrm{t}} e_2 : \tau \qquad \Delta \vdash_{\mathrm{k}} \tau : \mathrm{T}}{\Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{unpack}\ e_1\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ e_2 : \tau}$$

$$\text{ROLL}\ \frac{\Delta; \Gamma \vdash_{\mathrm{t}} e : \tau[\tau'/\beta][\mu(\sigma \leftarrow \beta)\alpha.\tau/\alpha] \qquad \Delta \vdash_{\mathrm{k}} \mu(\tau'{::}\sigma \leftarrow \beta)\alpha.\tau : \mathrm{T}}{\Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{roll}\ e\ \mathsf{as}\ \mu(\tau'{::}\sigma \leftarrow \beta)\alpha.\tau : \mu(\tau'{::}\sigma \leftarrow \beta)\alpha.\tau} \qquad \text{UNROLL}\ \frac{\Delta; \Gamma \vdash_{\mathrm{t}} e : \mu(\tau'{::}\sigma \leftarrow \beta)\alpha.\tau}{\Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{unroll}\ e : \tau[\tau'/\beta][\mu(\sigma \leftarrow \beta)\alpha.\tau/\alpha]}$$

$$\text{PEEL}\ \frac{\begin{array}{c}\Delta; \Gamma \vdash_{\mathrm{t}} e_1 : (\mu(\sigma \leftarrow \beta)\alpha.\tau_1) \times (\mu(\sigma \leftarrow \beta)\alpha.\tau_2) \\ \Delta, \alpha_{hd}{:}\mathrm{T}, \alpha_{tl}{:}\mathrm{L}; \Gamma, x{:}(\mu(\alpha_{hd}{::}\alpha_{tl} \leftarrow \beta)\alpha.\tau_1) \times (\mu(\alpha_{hd}{::}\alpha_{tl} \leftarrow \beta)\alpha.\tau_2) \vdash_{\mathrm{t}} e_2 : \tau \qquad \Delta \vdash_{\mathrm{k}} \tau : \mathrm{T}\end{array}}{\Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{peel}\ e_1\ \mathsf{as}\ \alpha_{hd}, \alpha_{tl}, x\ \mathsf{in}\ e_2 : \tau}$$

Figure 4: The typing rules for our coordinated-list language.

thus type-checking is straightforward.

3.4 An Extended Example

As in Section 2, we consider storing the environments and code pointers for a collection of closures separately. For brevity and readability, we use standard syntactic sugar such as type abbreviations, let $x{:}\tau = e_1$ in e_2 for $(\lambda x{:}\tau.\ e_2)\ e_1$, and let rec for fix. To emphasize that we do not restrict programs to use predefined data representations, we use different list encodings for the environments[3] and the code pointers[4]:

$$\begin{aligned}&\texttt{let t1} = \exists\beta'{:}\mathrm{L}.(\\ &\quad(\mu(\beta' \leftarrow \beta)\alpha.\ 1 + (\beta \times \alpha))\\ &\quad\times(\mu(\beta' \leftarrow \beta)\alpha.\ 1 + (\alpha \times ((\beta \times \mathsf{int}) \rightarrow \mathsf{int}))))\end{aligned}$$

The function `apply_nth` takes a `t1` and returns the application of 0 to the n^{th} closure, or 0 if the lists are too short:

```
let apply_nth : t1->int->int = λx. λ n.
 let rec f x n =
   peel x as α1,α2,x2 in
   case unroll(π1 x2) of
   x3. 0 (*1st list too short*)
   x3. case unroll(π2 x2) of
     x4. 0 (*2nd list too short*)
     x4. if n==1(*apply closure or recur*)
         then (π2 x4) ((π1 x3), 0)
         else f ((π2 x3), (π1 x4)) (n-1)
 in
 unpack x as β',x1 in
 f x1 n
```

Without the peel coercion, the subsequent unroll expression would not typecheck because π_1 `x` has a type of the form $\mu(\beta' \leftarrow \beta)\alpha.\tau$. Furthermore, we must peel both components of `x` simultaneously or the function application in the then branch would not type-check.

Adding a closure to a list involves creating an existential type abstracting a larger list of types:

```
let cons: t1→ (∃α.α × ((α × int) → int)) →t1
= λ x. λ c.
   unpack x as β',x2 in
   unpack c as α',c2 in
   pack α'::β',
   ((roll (in2 (π1 c2, π1 x2)) as
     μ(α'::β' ← β)α. 1 + (β × α)),
    (roll (in2 (π2 x2, π2 c2)) as
     μ(α'::β' ← β)α. 1 + (α × ((β × int) → int))))
    as t1
```

[3]We place the next-node pointer at the end of each node.
[4]We place the next-node pointer at the front of each node.

One problem remains: How do we make the pair of empty lists we need to represent that there are no closures? For the recursive types, any σ suffices, so we can use 1^*. Infinite lists of types ensure the peel coercion in `apply_nth` never gets stuck.

```
let empty_list : t1 =
  pack 1*,
   ((roll (in1 ()) as
     μ(1* ← β)α. 1 + (β × α)),
    (roll (in1 ()) as
     μ(1* ← β)α. 1 + (α × ((β × int) → int))))
  as t1
```

The encoding of an empty list appears daunting, but it needs to be done only once.

4 Metatheory

This section describes the important metatheoretic results we established for the language presented in Section 3. In Section 4.1, we outline our type-safety proof. In Section 4.2, we briefly describe a set of type-erasure rules and a corresponding type-erasure theorem.

4.1 Type Safety

We now sketch a proof of type safety for our coordinated list language. The accompanying technical report [21] contains detailed proofs of all lemmas. We consider the most interesting cases here.

Definition 1 (Stuck)

An expression e is stuck if e is not a value and there is no e' such that $e \rightarrow e'$.

Theorem 2 (Type Safety)

If $\cdot;\cdot \vdash_{\mathrm{t}} e : \tau$ and $e \rightarrow^* e'$ (where $\rightarrow^*$ is the reflexive, transitive closure of $\rightarrow$), then e' is not stuck.

Proof sketch: As usual, type safety is a corollary of the Preservation and Progress Lemmas [25].

Lemma 3 (Preservation)

1. *If $\cdot;\cdot \vdash_{\mathrm{t}} e : \tau$ and $e \xrightarrow{\mathrm{r}} e'$, then $\cdot;\cdot \vdash_{\mathrm{t}} e' : \tau$.*
2. *If $\cdot;\cdot \vdash_{\mathrm{t}} e : \tau$ and $e \rightarrow e'$, then $\cdot;\cdot \vdash_{\mathrm{t}} e' : \tau$.*

Proof sketch: We consider only the first lemma, as the second lemma is a corollary. Our proof is by cases on the reduction rules. We present the peel reduction case here:

- **Case** $e =$
  $\mathsf{peel}(\mathsf{roll}\ v_1\ \mathsf{as}\ \mu(\sigma \leftarrow \beta)\alpha.\tau_1,$
  $\quad \mathsf{roll}\ v_2\ \mathsf{as}\ \mu(\sigma \leftarrow \beta)\alpha.\tau_2)$
  $\mathsf{as}\ \alpha_{hd}, \alpha_{tl}, x\ \mathsf{in}\ e_1 \xrightarrow{\mathrm{r}} e'$
  where $e' =$
  $e_1[\tau'/\alpha_{hd}][\sigma'/\alpha_{tl}]$
  $\quad[(\mathsf{roll}\ v_1\ \mathsf{as}\ \mu(\tau'::\sigma' \leftarrow \beta)\alpha.\tau_1,$
  $\quad\ \mathsf{roll}\ v_2\ \mathsf{as}\ \mu(\tau'::\sigma' \leftarrow \beta)\alpha.\tau_2)/x]$
  and $\mathrm{peel}(\sigma) = \tau'::\sigma'$**:**
  By the PEEL typing rule, $\cdot;\cdot \vdash_{\mathrm{t}} e : \tau$ ensures:

  (1) $\cdot \vdash_{\mathrm{k}} \tau : \mathrm{T}$

  (2)
  $$\cdot;\cdot \ \vdash_{\mathrm{t}}\ (\mathsf{roll}\ v_1\ \mathsf{as}\ \mu(\sigma \leftarrow \beta)\alpha.\tau_1,\ \mathsf{roll}\ v_2\ \mathsf{as}\ \mu(\sigma \leftarrow \beta)\alpha.\tau_2) : \mu(\sigma \leftarrow \beta)\alpha.\tau_1 \times \mu(\sigma \leftarrow \beta)\alpha.\tau_2$$

  (3)
  $$\cdot, \alpha_{hd}{:}\mathrm{T}, \alpha_{tl}{:}\mathrm{L}; \cdot, x{:}(\mu(\alpha_{hd}::\alpha_{tl} \leftarrow \beta)\alpha.\tau_1) \times (\mu(\alpha_{hd}::\alpha_{tl} \leftarrow \beta)\alpha.\tau_2) \vdash_{\mathrm{t}} e_1{:}\tau$$

  By inversion of (1) and the peel metafunction, $\cdot \vdash_{\mathrm{k}} \tau' : \mathrm{T}$ and $\cdot \vdash_{\mathrm{k}} \sigma' : \mathrm{L}$. With this, (2), and (3), the Substitution Lemma (below) can conclude $\cdot;\cdot \vdash_{\mathrm{t}} e' : \tau$.

Lemma 4 (Progress)

1. *If $\cdot;\cdot \vdash_{\mathrm{t}} e : \tau$ and e is not a value then there exists an E, e_r, and e'_r such that $e = E[e_r]$ and $e_r \xrightarrow{\mathrm{r}} e'_r$.*
2. *If $\cdot;\cdot \vdash_{\mathrm{t}} e : \tau$ then e is a value or there exists an e' such that $e \to e'$.*

Proof sketch: Again, we consider only the first lemma. The proof is by induction on the structure of e. We consider the peel case:

- If e is some $\mathsf{peel}\ e_1\ \mathsf{as}\ \alpha_{hd}, \alpha_{tl}, x\ \mathsf{in}\ e_2$, then inverting $\cdot;\cdot \vdash_{\mathrm{t}} e : \tau$ ensures that $\cdot;\cdot \vdash_{\mathrm{t}} e_1 : \mu(\sigma \leftarrow \beta)\alpha.\tau_1 \times \mu(\sigma \leftarrow \beta)\alpha.\tau_2$. If e_1 is not a value, then by induction there are E_1 and e_r such that $e_1 = E_1[e_r]$ and $e_r \xrightarrow{\mathrm{r}} e'_r$. Then $e = \mathsf{peel}\ E_1[e_r]\ \mathsf{as}\ \alpha_{hd}, \alpha_{tl}, x\ \mathsf{in}\ e_2$, so letting $E = \mathsf{peel}\ E_1\ \mathsf{as}\ \alpha_{hd}, \alpha_{tl}, x\ \mathsf{in}\ e_2$ suffices. Otherwise, if e_1 is a value then the canonical forms of pair types and recursive types (and inversion of the PAIR rule), ensures that e_1 has the form $(\mathsf{roll}\ v_1\ \mathsf{as}\ \mu(\sigma \leftarrow \beta)\alpha.\tau_1, \mathsf{roll}\ v_2\ \mathsf{as}\ \mu(\sigma \leftarrow \beta)\alpha.\tau_2)$.
  Thus $e \xrightarrow{\mathrm{r}} e_2[\tau'/\alpha_{hd}][\sigma'/\alpha_{tl}]$
  $\quad[(\mathsf{roll}\ v_1\ \mathsf{as}\ \mu(\tau'::\sigma' \leftarrow \beta)\alpha.\tau_1,$
  $\quad\ \mathsf{roll}\ v_2\ \mathsf{as}\ \mu(\tau'::\sigma' \leftarrow \beta)\alpha.\tau_2)/x]$
  where $\mathrm{peel}(\sigma) = \tau'::\sigma'$. Thus $[\cdot]$ suffices for E.

Lemma 5 (Substitution)

1. *If $\Delta, \alpha{:}\kappa'; \Gamma \vdash_{\mathrm{t}} e : \tau$ and $\Delta \vdash_{\mathrm{k}} \tau' : \kappa'$, then $\Delta; \Gamma[\tau'/\alpha] \vdash_{\mathrm{t}} e[\tau'/\alpha] : \tau[\tau'/\alpha]$.*
2. *If $\Delta; \Gamma, x{:}\tau' \vdash_{\mathrm{t}} e : \tau$ and $\Delta; \Gamma \vdash_{\mathrm{t}} e' : \tau'$, then $\Delta; \Gamma \vdash_{\mathrm{t}} e[e'/x] : \tau$.*

Proof sketch: By induction on the typing derivations for e.

4.2 Erasure

We define an `erase` metafunction that converts expressions in our typed language into equivalent expressions in an untyped language. The erasure rules for our language are all standard, and can be found in the accompanying technical report [21]. The rule for `peel` is typical for coercions:

$$\mathtt{erase}(\mathsf{peel}\ e_1\ \mathsf{as}\ \alpha_{hd}, \alpha_{tl}, x\ \mathsf{in}\ e_2) = (\lambda x.\ \mathtt{erase}(e_2))\ \mathtt{erase}(e_1)$$

The technical report proves erasure and evaluation commute:

Theorem 6 (Erasure Theorem)

If e is an expression in the typed language, v is a value in the typed language, and $e \to^\star v$, then $\mathtt{erase}(e) \to^\star \mathtt{erase}(v)$ in the untyped language. (Also, e and $\mathtt{erase}(e)$ have the same termination behavior.)

5 Encoding Conventional Recursive Types

This section considers whether we can replace conventional recursive types (of the form $\mu\alpha.\tau$) with our enriched recursive types (of the form $\mu(\sigma \leftarrow \beta)\alpha.\tau$). It turns out to require a richer theory of type equality.

There is no problem with a type system supporting both forms of recursive types separately. We simply need separate roll/unroll coercions and typing rules for the distinct forms.

Nonetheless, minimalist type systems are appealing. They reduce the metatheoretic proof obligations and ease implementation. It also helps to understand one form of type by considering if it is strictly more expressive than another. Toward this end, we consider why it is difficult to encode conventional recursive types with our enriched types (Section 5.1). We then consider two possible solutions. The first, a subtyping-style approach (Section 5.2), solves the problem but requires extensions to the language. The other, a type abstraction approach (Section 5.3), seems intriguing but unfortunately fails to solve the problem.

5.1 The Problem

Intuitively, we expect any encoding to translate $\mu\alpha.\tau$ to some $\mu(\sigma \leftarrow \beta)\alpha.\tau'$ where β does not appear free in τ'. For simplicity, it is important for the translation to be injective in the sense that all expressions of the same source type are translated to expressions of the same target type. Otherwise, we have to mediate type mismatches. For example, we do not want the translation of `if` e_1 `then` e_2 `else` e_3 to give e_2 and e_3 different types.

Because β is unused, we just need a σ of kind L, and 1^* seems like a fine choice. Unfortunately, our typing rule for roll is insufficient. To see why, consider this pre-encoding cons function for an ordinary list of integers:

```
let cons (i:int) (lst:μα.1 + int × α) =
   roll (in2 (i,lst)) as μα.1 + int × α
```

A straightforward translation may look like this:

```
let cons (i:int) (lst:μ(1*←β)α.1 + int×α) =
   roll (in2 (i,lst)) as
     μ(1::1* ← β)α.1 + int×α
```

The typing rule for roll must give a roll expression a type of the form $\mu(\tau :: \sigma \leftarrow \beta)\alpha.\tau'$. This translation of cons is incorrect; its result type is not the translated type of integer lists.

Conversely, the type $\mu(1^* \leftarrow \beta)\alpha.1 + \mathsf{int} \times \alpha$ does not let us unroll a value of the type. We can use a peel coercion and then an unroll, but peel abstracts 1^* with some α_{hd} and α_{tl}. Hence we can give the tail of a linked list a type like $\mu(\alpha_{tl} \leftarrow \beta)\alpha.1 + \mathsf{int} \times \alpha$, but not $\mu(1^* \leftarrow \beta)\alpha.1 + \mathsf{int} \times \alpha$.

5.2 The Subtyping Approach

We can add coercions to do what we need:

$$\frac{\Delta;\Gamma \vdash_\tau e : \mu(\tau_1::{\tau_1}^* \leftarrow \beta)\alpha.\tau_2}{\Delta;\Gamma \vdash_\tau \mathsf{shorten}\ e : \mu({\tau_1}^* \leftarrow \beta)\alpha.\tau_2}$$

$$\frac{\Delta;\Gamma \vdash_\tau e : \mu({\tau_1}^* \leftarrow \beta)\alpha.\tau_2}{\Delta;\Gamma \vdash_\tau \mathsf{lengthen}\ e : \mu(\tau_1::{\tau_1}^* \leftarrow \beta)\alpha.\tau_2}$$

Similarly, if we had subtyping (or type equivalence) for type lists,[5] $\tau^* \leq \tau::\tau^*$ and $\tau::\tau^* \leq \tau^*$, we would not need explicit coercions. For languages with subtyping (or type equivalence), this solution works well and encodes an obvious equivalence. Otherwise, adding a theory of equality may be more trouble than simply using two forms of recursive types.

[5]The polarity of a type list σ in a recursive type $\mu(\sigma \leftarrow \beta)\alpha.\tau$ depends on the polarity of the free βs in τ. If every β appears in a covariant context, then σ is covariant. If every β appears in a contravariant context, then σ is contravariant. Otherwise σ is invariant.

5.3 The Type Abstraction Approach

Alternatively, to avoid mismatches between "placeholder" types (such as 1^* and $1::1^*$), we can try to translate conventional recursive types into existential packages that hide the type lists; i.e., translate $\mu\alpha.\tau$ to $\exists\beta'{:}\mathrm{L}.\mu(\beta' \leftarrow \beta)\alpha.\tau$ (assuming τ has no types of the form $\mu\alpha'.\tau'$).

At the term level, this type translation requires an unroll coercion to become an unpack (to remove the existential quantifier), a peel[6] (to separate the newly unpacked type list into a head and a tail), and an unroll, which is fine: None of these coercions have a run-time effect.

But no translation of the roll coercion works. We would like to translate roll to a roll followed by a pack, but the roll coercion will not type-check when applied to a translated expression. For example, in_2 `(i,lst)` in the body of cons can have type $1 + \mathsf{int} \times \exists\beta'{:}\mathrm{L}.\mu(\beta' \leftarrow \beta)\alpha.1 + \mathsf{int} \times \alpha$. The roll coercion cannot apply because it would require using β' outside of its scope, producing an ill-formed type. In this example, we can write cons by unpacking `lst` and using the unpacked version to build `(i,lst)`, which corresponds to the implementation of cons in Section 3.4.

However, this approach is awkward to automate fully. Roughly, to perform a roll, we need a version of the rolled-expression where occurrences of the type being rolled have all been unpacked. Creating this version may require making a copy (e.g., if the untranslated expression is a variable). Making this copy has run-time cost and requires different code for each recursive type being encoded.

Alternately, we might devise a "deep coercion" for treating arbitrary data structures as though some of their existential types are unpacked. Such a coercion would be difficult to define and would almost surely be unsound in the presence of mutation [8].

6 Synergy With Singleton-Integer Types

In this section, we discuss the synergy between our extensions and singleton integers. We show how the combination of the two lets us create a pair of lists where only one list has tags.

The standard implementation of typed recursive data structures requires that each node include a run-time *tag* indicating whether or not the node has recursive children. For instance, in a list with type

[6]Technically peel needs a pair, but we can peel a pair with identical components (e.g., $\mathsf{peel}\ (y, y)\ \mathsf{as}\ \alpha_{hd}, \alpha_{tl}, x\ \mathsf{in}\ e_2$) and then unroll the first component (e.g., $\pi_1\ x$).

$$
\begin{array}{rcll}
\kappa & ::= & \ldots \mid \mathrm{I} & \\
\tau & ::= & \ldots \mid i \mid \mathrm{S}(\tau) \mid \mathsf{if}\ \tau\ \mathsf{then}\ \tau\ \mathsf{else}\ \tau & i \in \mathcal{Z} \\
e & ::= & \ldots \mid i \mid \mathsf{tosum}\ e, \tau \mid \mathsf{match}\ e\ x.e\ x.e & i \in \mathcal{Z} \\
v & ::= & \ldots \mid i \mid \mathsf{tosum}\ v, \tau & i \in \mathcal{Z} \\
E & ::= & \ldots \mid \mathsf{tosum}\ E, \tau \mid \mathsf{match}\ E\ x.e\ x.e &
\end{array}
\qquad \mathsf{match}\ (i, (\mathsf{tosum}\ v, E))\ x.e_1\ x.e_0 \xrightarrow{\mathrm{r}} e_i[v/x]
$$

$$
\frac{i \in \mathcal{Z}}{\Delta \vdash_{\mathrm{k}} i : \mathrm{I}}
\qquad
\frac{\Delta \vdash_{\mathrm{k}} \tau : \mathrm{I}}{\Delta \vdash_{\mathrm{k}} \mathrm{S}(\tau) : \mathrm{T}}
\qquad
\frac{i \in \mathcal{Z}}{\Delta; \Gamma \vdash_{\mathrm{t}} i : \mathrm{S}(i)}
$$

$$
\frac{\Delta; \Gamma \vdash_{\mathrm{t}} e_1 : \tau_1 \times \mathsf{if}\ \tau_1\ \mathsf{then}\ \tau_2\ \mathsf{else}\ \tau_3 \qquad \Delta; \Gamma, x{:}\tau_2 \vdash_{\mathrm{t}} e_2 : \tau_4 \qquad \Delta; \Gamma, x{:}\tau_3 \vdash_{\mathrm{t}} e_3 : \tau_4}{\Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{match}\ e_1\ x.e_2\ x.e_3 : \tau_4}
$$

$$
\frac{\Delta; \Gamma \vdash_{\mathrm{t}} e : \tau_i \qquad \Delta \vdash_{\mathrm{k}} \tau_{1-i} : \mathrm{T} \qquad i \in \{0, 1\}}{\Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{tosum}\ e, \mathsf{if}\ \mathrm{S}(i)\ \mathsf{then}\ \tau_1\ \mathsf{else}\ \tau_0 : \mathsf{if}\ \mathrm{S}(i)\ \mathsf{then}\ \tau_1\ \mathsf{else}\ \tau_0}
$$

Figure 5: The extensions for singleton integers and conditional types.

$\mu\alpha.1 + \mathsf{int} \times \alpha$, each node carries a tag indicating whether it has type 1 or type $\mathsf{int} \times \alpha$. These tags seem necessary to discriminate between node types.

We can partially eliminate tags, though, by combining our enriched recursive types with *singleton integers* and *conditional types* (see, e.g., [1]). Specifically, we can create a pair of coordinated lists[7] where one list has tagged nodes and the other has tagless nodes. The type of the coordinated pair ensures that corresponding nodes in the two lists are either both empty or both non-empty. That is, both lists have the same length. The type also ensures that a node of the tagless list cannot be accessed without first checking the tag of the corresponding node in the tagged list. Neither singleton integers nor conditional types are new—it is their combination with our enriched recursive types that enables this encoding.

We first extend our language to include singleton integers. Figure 5 contains the necessary changes. We add a new integer kind (I), a new expression form for integers (i), and two new type forms (i and $\mathrm{S}(\tau)$). We use $\mathcal{Z}$ to denote the set of integers. To avoid ambiguity, we now use unit for the unit type, rather than 1. We also add three new typing rules. The rules state that integer types (i) have kind I, singleton integer types ($\mathrm{S}(\tau)$, where τ has kind I) have kind T, and that each integer expression i has singleton type $\mathrm{S}(i)$. A type for all integers is just $\mathsf{int} = \exists\alpha{:}\mathrm{I}.\mathrm{S}(\alpha)$.

Figure 5 also contains the extensions for conditional types. We require a new conditional type ($\mathsf{if}\ \tau\ \mathsf{then}\ \tau\ \mathsf{else}\ \tau$), two new expression forms for constructing and branching on conditional types ($\mathsf{tosum}\ e, \tau$ and $\mathsf{match}\ e\ x.e\ x.e$), and a new value form ($\mathsf{tosum}\ v, \tau$). We also add new typing rules, expression contexts, and reductions for tosum and match. The semantics and typing rules show that match, tosum, and conditional types can achieve the same effect as case and sum types, but with more control over the data representation (see, for instance, [15] and [26]).

These additions let us create a coordinated pair consisting of a tagged and a tagless list. We give the pair the type:

$$
\begin{array}{l}
\mathsf{tagged\_tagless} = \exists\beta'{:}\mathrm{L}. \\
\quad ((\mu(\beta' \leftarrow \beta)\alpha.(\beta \times \mathsf{if}\ \beta\ \mathsf{then}\ \mathsf{unit}\ \mathsf{else}\ (\mathsf{int} \times \alpha))) \times \\
\quad\ (\mu(\beta' \leftarrow \beta)\alpha.\qquad \mathsf{if}\ \beta\ \mathsf{then}\ \mathsf{unit}\ \mathsf{else}\ (\mathsf{int} \times \alpha)))
\end{array}
$$

The first list is tagged (with β), and the second list is untagged. However, the conditional types of both lists depend on β. The two lists are coordinated, so both β are drawn from the same type list β'. Thus, for each pair of corresponding nodes, the conditional types must evaluate to the same branch. That is, both will evaluate to unit (an empty list), or both will evaluate to $\mathsf{int} \times \alpha$ (a non-empty list). Figure 6 presents a picture of this data structure.

We also know that all accesses to the untagged list must first check the corresponding tag in the tagged list. Recall that conditionally typed values can be accessed only with match expressions. For the match to typecheck, it must be passed a pair with type of the form $\mathrm{S}(i) \times \mathsf{if}\ \mathrm{S}(i)\ \mathsf{then}\ \tau_1\ \mathsf{else}\ \tau_2$. For the tagless list, the if-clause type (τ_1 in $\mathsf{if}\ \tau_1\ \mathsf{then}\ \tau_2\ \mathsf{else}\ \tau_3$) is drawn from the existentially quantified list β'. Thus the *only* integer which can be used as the first element of the pair is the only integer known to have the same type: the tag of the corresponding element of the tagged list.

[7] We can do the same thing for general recursive data structures using the extensions described in Section 7.

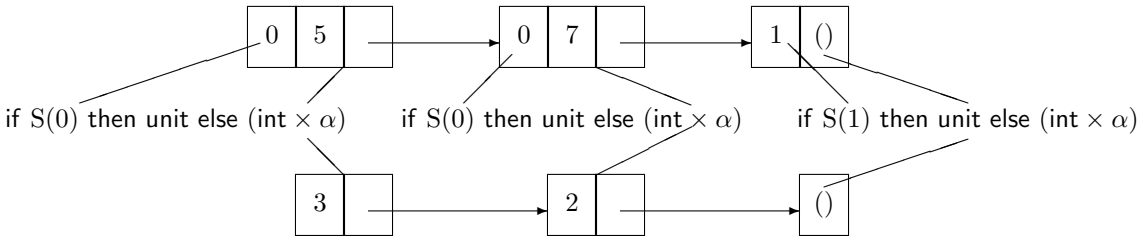

Figure 6: A tagged-tagless pair of lists. The type of the tags and the conditions of the if-then-else types are drawn from an existentially quantified type list σ that is shared by both lists.

We have written `zip` and `unzip` functions for tagged-tagless pairs of lists (see our website [28]). The `zip` function converts a tagged-tagless pair into a single list of pairs, and `unzip` is its inverse.

7 A Full Language Supporting General Recursive Data Structures

In this section, we generalize our enriched recursive types to support recursive data structures with multiple children (e.g., trees). We use this extension to encode a red-black tree [4] implementation of integer sets as a pair of trees, where one tree contains the values and the other tree contains the colors. Our enhanced type system guarantees that the two trees have the same shape: Every value node has a corresponding color node, and vice versa. By splitting the tree like this, the lookup function needs to access only the value tree. In some cases, this may lead to better cache performance. Similarly, if our red-black tree were a dictionary (with a key and value for each conceptual node), separating the keys and the values could make functions accessing only the keys (e.g., "is member") faster.

7.1 The Full Language

The enriched recursive types presented in Section 3 are not well suited for encoding coordinated sets of recursive data structures with more than one child. For example, consider a pair of coordinated binary trees. A conventional encoding of a binary tree of pairs will have a type similar to

$$\mu\alpha.\mathsf{unit} + ((\exists\beta.(\beta \times \beta)) \times (\alpha \times \alpha)) \,.$$

If we attempt to encode this tree as a coordinated pair of trees with our enriched recursive types, we will end up with a type of the form

$$\begin{aligned}\exists\sigma{:}\mathrm{L}.((\mu(\sigma \leftarrow \beta)\alpha.\mathsf{unit} + (\beta \times (\alpha \times \alpha))) \times \\ (\mu(\sigma \leftarrow \beta)\alpha.\mathsf{unit} + (\beta \times (\alpha \times \alpha)))) \,.\end{aligned}$$

When we peel this pair, we get back a single α_{tl} list tail. When we unroll one of the trees, the same tail list will be used for *both* children. So each child will share types drawn from σ not only with the corresponding child in the *other* tree, but also with its sibling in the *same* tree. In other words, if node A_1 in tree A has children A_2 and A_3, and corresponding node B_1 in tree B has children B_2 and B_3, then the four nodes A_2, A_3, B_2, and B_3 all share types with each other. This result is unsatisfactory if we only want A_2 to share with B_2, and A_3 to share with B_3.

We solve this problem by replacing our *type lists* with *type trees*. Type trees are like type lists, except that each tree has n children instead of a single tail. For generality, we also add multiple types to each node of the type tree, instead of a single head. Multiple types allow coordinated nodes to share multiple existential types.

We describe the syntactic modifications in Figure 7. We have a new kind $\mathrm{L}^{(m,n)}$ for n-ary type trees with m coordinated types per node. For example, our red-black tree example will use type trees of kind $\mathrm{L}^{(1,2)}$. We also modify our recursive types to take m βs (one for each of the coordinated types) and n αs (one for each of the type tree's children). The modified star type ^{*n} takes a sequence τ^m of m conventional types and generates an infinite n-ary tree of nodes containing the types in τ^m. The modified cons type $::^{m,n}$ takes a sequence τ^m of m conventional types and a sequence σ^n of n type trees of kind $\mathrm{L}^{(m,n)}$, and returns a tree whose root contains the types in τ^m and whose children are the trees in σ^n. We also modify the syntax of peel to take $m+n$ type variables—one for each of the m coordinated types and one for each of the n children.

Figure 7 also describes the necessary semantic modifications for this generalization. Only the peel reduction and peel metafunction change. Applying the peel metafunction to a type tree constructed with $::^{m,n}$ yields the same tree (as was the case

$$\text{let } \tau^m = \tau_1, \tau_2, \ldots, \tau_m$$

$$\begin{array}{rcl} \kappa & ::= & \ldots \mid \mathrm{L}^{(m,n)} \\ \sigma, \tau & ::= & \ldots \mid \mu(\sigma \leftarrow (\beta^m))(\alpha^n).\tau \mid (\tau^m)^{*n} \mid (\tau^m)::^{m,n}(\sigma^n) \\ e & ::= & \ldots \mid \mathsf{peel}\ e\ \mathsf{as}\ \alpha^m, \beta^n, x\ \mathsf{in}\ e \\ E & ::= & \ldots \mid \mathsf{peel}\ E\ \mathsf{as}\ \alpha^m, \beta^n, x\ \mathsf{in}\ e \end{array}$$

$$\begin{array}{l} \mathsf{peel}(\mathsf{roll}\ v_1\ \mathsf{as}\ \mu(\sigma \leftarrow \beta^m)\alpha^n.\tau_1, \\ \quad \mathsf{roll}\ v_2\ \mathsf{as}\ \mu(\sigma \leftarrow \beta^m)\alpha^n.\tau_2)\ \mathsf{as}\ \alpha^m_{hd}, \alpha^n_{tl}, x\ \mathsf{in}\ e \xrightarrow{\mathrm{r}} \\ e[\tau'_i/\alpha_{hd,i}][\sigma'_j/\alpha_{tl,j}][(\mathsf{roll}\ v_1\ \mathsf{as}\ \mu((\tau'^m)::^{m,n}(\sigma'^n) \leftarrow (\beta^m))(\alpha^n).\tau_1, \\ \qquad\qquad \mathsf{roll}\ v_2\ \mathsf{as}\ \mu((\tau'^m)::^{m,n}(\sigma'^n) \leftarrow (\beta^m))(\alpha^n).\tau_2)/x] \\ \text{for all } i \in [1,m] \text{ and } j \in [1,n], \text{ where } \mathrm{peel}(\sigma) = (\tau'^m)::^{m,n}(\sigma'^n) \end{array}$$

$$\begin{array}{rcl} \mathrm{peel}((\tau^m)::^{m,n}(\sigma^n)) & = & (\tau^m)::^{m,n}(\sigma^n) \\ \mathrm{peel}((\tau^m)^{*n}) & = & (\tau^m)::^{m,n}(((\tau^m)^{*n})^n) \end{array}$$

Figure 7: The syntactic and semantic modifications for the full coordinated data structure language.

with type lists constructed with ::). Applying the peel metafunction to a type tree constructed with $(\tau^m)^{*n}$ yields a tree with a root containing the conventional types in τ^m, and with n copies of $(\tau^m)^{*n}$ as children. The modified peel reduction is simply the type tree analog of the previously described peel reduction for type lists. The original peel reduction substituted the head of the type list (τ' in Figure 3) for α_{hd}. The new peel instead substitutes each element τ'_i of the type tree head for the corresponding type variable $\alpha_{hd,i}$. Similarly, where the original peel substituted the list tail σ' for α_{tl}, the new peel substitutes each child tree σ'_j for the corresponding type variable $\alpha_{tl,j}$.

We describe the modified typing rules in Figure 8. The KMU, KSTAR and KCONS rules simply formalize what we described above, and ensure that the kinds and type variables match up with the number of coordinated types and children. The type tree versions of the ROLL, UNROLL, and PEEL rules are identical to their type list versions, except that we now substitute all m coordinated types and all n children.

7.2 Split Red-Black Trees

In this section, we describe an implementation of red-black trees in our full language. Our implementation uses our enriched recursive types to encode the tree as two separate but coordinated trees. The first tree is tagged and contains the values, and the second tree is tagless and contains the corresponding colors. As was the case in Section 6, the type of the pair guarantees that the two trees have the same shape. A visual representation of this encoding is shown in Figure 9. Because lookups access only the values of each node, we do not need to use the color tree unless we are adding or removing nodes. Our encoding has type rbtree =

$$\begin{array}{l} \exists\beta'{:}\mathrm{L}^{(1,2)}. \\ \quad (\mu(\beta' \leftarrow (\beta))(\alpha_1, \alpha_2). \\ \qquad (\beta \times \mathsf{if}\ \beta\ \mathsf{then\ unit\ else}\ (\mathsf{int} \times (\alpha_1 \times \alpha_2))) \times \\ \quad \mu(\beta' \leftarrow (\beta))(\alpha_1, \alpha_2). \\ \qquad (\mathsf{if}\ \beta\ \mathsf{then\ black\_t\ else}\ (\mathsf{color} \times (\alpha_1 \times \alpha_2)))) \end{array}$$

where color = int, (black = 0 and red = 1), and black_t = S(0). The empty leaf nodes of red-black trees are always colored black, thus the then-clause of the second tree's conditional type is black_t—the type of the value black.

Our website [28] contains lookup and insert functions implemented for red-black tree-pairs. The recursive procedure in lookup traverses only the value-tree; it does not even have a color-tree argument.

8 Implementation

We implemented an interpreter for our language in O'Caml. We verified that the examples in previous sections typecheck and evaluate as expected. We also confirmed that preservation and type erasure hold during evaluation. The interpreter and examples can be found on our website [28].

Our red-black tree implementation contains code for an empty tree, a lookup function, and an insert function. Lookup takes an integer and a tree pair. The value tree component of the pair is passed to another function which recursively searches for the key. Insert takes a tree pair and an integer, inserts the integer, and balances the tree pair.

KMU

$$\frac{\Delta \vdash_{\mathrm{k}} \sigma : \mathrm{L}^{(m,n)} \qquad \Delta, \alpha_i{:}\mathrm{T}, \beta_j{:}\mathrm{T} \vdash_{\mathrm{k}} \tau : \mathrm{T},\ \forall i \in [1,n],\ \forall j \in [1,m]}{\Delta \vdash_{\mathrm{k}} \mu(\sigma \leftarrow (\beta^m))(\alpha^n).\tau : \mathrm{T}}$$

KSTAR

$$\frac{\Delta \vdash_{\mathrm{k}} \tau_i : \mathrm{T},\ \forall i \in [1,m]}{\Delta \vdash_{\mathrm{k}} (\tau^m)^{*n} : \mathrm{L}^{(m,n)}}$$

KCONS

$$\frac{\Delta \vdash_{\mathrm{k}} \tau_i : \mathrm{T},\ \forall i \in [1,m] \qquad \Delta \vdash_{\mathrm{k}} \sigma_i : \mathrm{L}^{(m,n)},\ \forall i \in [1,n]}{\Delta \vdash_{\mathrm{k}} (\tau^m)::^{m,n}(\sigma^n) : \mathrm{L}^{(m,n)}}$$

ROLL

$$\frac{\begin{array}{c}\Delta \vdash_{\mathrm{k}} \mu((\tau'^m)::^{m,n}(\sigma^n) \leftarrow (\beta^m))(\alpha^n).\tau : \mathrm{T} \\ \Delta; \Gamma \vdash_{\mathrm{t}} e : \tau[\tau'_j/\beta_j][\mu(\sigma_i \leftarrow (\beta^m))(\alpha^n).\tau/\alpha_i],\ \forall i \in [1,n],\ \forall j \in [1,m]\end{array}}{\Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{roll}\ e\ \mathsf{as}\ \mu((\tau'^m)::^{m,n}(\sigma^n) \leftarrow (\beta^m))(\alpha^n).\tau : \mu((\tau'^m)::^{m,n}(\sigma^n) \leftarrow (\beta^m))(\alpha^n).\tau}$$

UNROLL

$$\frac{\Delta; \Gamma \vdash_{\mathrm{t}} e : \mu((\tau'^m)::^{m,n}(\sigma^n) \leftarrow (\beta^m))(\alpha^n).\tau}{\Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{unroll}\ e : \tau[\tau'_j/\beta_j][\mu(\sigma_i \leftarrow (\beta^m))(\alpha^n).\tau/\alpha_i],\ \forall i \in [1,n],\ \forall j \in [1,m]}$$

PEEL

$$\frac{\begin{array}{c}\tau_{\mathrm{pair},i} = \mu(\alpha^m_{hd}::\alpha^n_{tl} \leftarrow \beta^m)\alpha^n.\tau_i \qquad \Delta; \Gamma \vdash_{\mathrm{t}} e_1 : (\mu(\sigma \leftarrow \beta^m)\alpha^n.\tau_1) \times (\mu(\sigma \leftarrow \beta^m)\alpha^n.\tau_2) \\ \Delta, \alpha_{hd,j}{:}\mathrm{T}, \alpha_{tl,i}{:}\mathrm{L}; \Gamma, x{:}\tau_{\mathrm{pair},1} \times \tau_{\mathrm{pair},2} \vdash_{\mathrm{t}} e_2 : \tau,\ \forall i \in [1,n],\ \forall j \in [1,m] \qquad \Delta \vdash_{\mathrm{k}} \tau : \mathrm{T}\end{array}}{\Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{peel}\ e_1\ \mathsf{as}\ \alpha^m_{hd}, \alpha^n_{tl}, x\ \mathsf{in}\ e_2 : \tau}$$

Figure 8: The modified typing rules for the full coordinated data structure language.

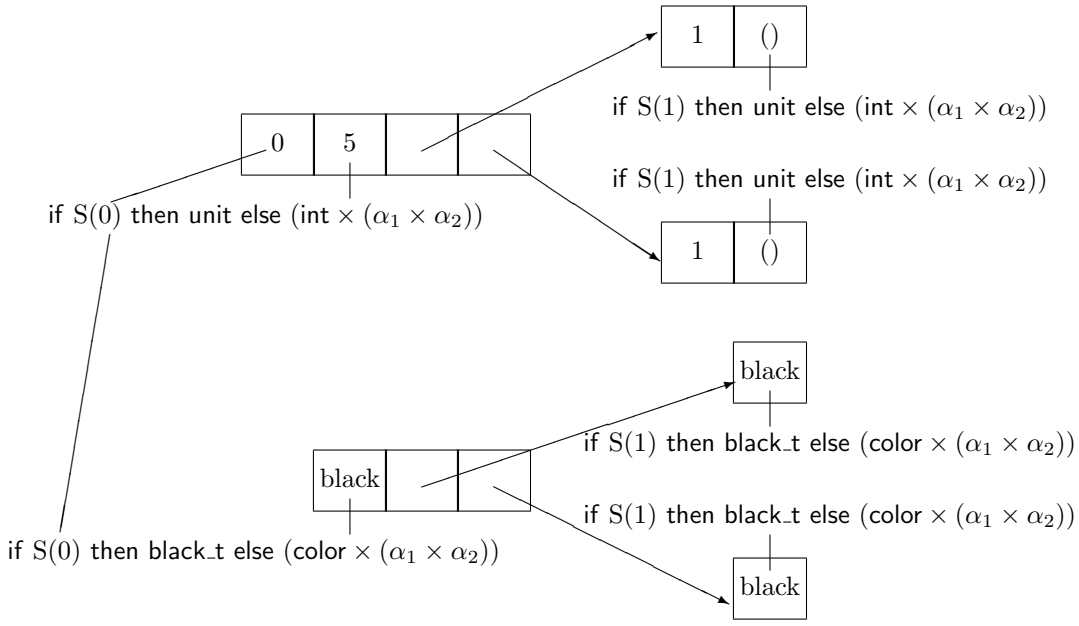

Figure 9: A red-black tree encoded as a pair of coordinated trees. The top tree contains the tags and values, and the bottom tree contains the corresponding colors.

9 Related Work

Typed assembly languages and proof-carrying code frameworks (e.g, [16, 18, 12, 2, 3, 5]) aim to provide expressive type languages so that compilers can choose natural and efficient data representations. To our knowledge, none of these systems support coordinated data structures. Many do have singleton types (which have many uses such as enforcing lock-based mutual exclusion [7, 9] or region-based memory management [23, 11]), so our work could make them more useful.

Crary and Weirich's formal language LX [6] can actually encode coordinated data structures even though LX was designed for flexible runtime type analysis. Very roughly, (1) parameterized recursive types can encode $\mu(\sigma \leftarrow \beta)\alpha.\tau$ as $\texttt{rec}((\lambda\alpha.\lambda\beta.\tau),\sigma)$, inductive kinds and pair kinds can encode $\mathrm{L}^{(m,n)}$, and primitive recursion [13] can encode the peel coercion. In this sense, our technical contribution is finding a much *less* powerful language that is powerful enough for our purposes. (LX essentially provides a rich but strongly-normalizing programming language at the type level.) Our simplicity better demonstrates what is necessary for coordinated data structures and makes it more likely that techniques for efficient type-checkers (e.g., hash-consing [22, 10] and explicit substitutions [17]) will apply. Conversely, we have demonstrated some of what LX can do; prior work did not consider coordinated data structures or provide examples of them in a typed language.

Xi's work on dependent types [27, 26] has expressiveness that overlaps with our work, but it is actually incomparable. Both approaches can enforce that two lists of unknown length have the same length. By using type-level arithmetic, dependent types can also enforce that an append function returns a list of length $n + m$ given lists of lengths n and m. But arithmetic summarizes quite a bit; it cannot express that corresponding elements of two lists are related. Similarly, Xi's dependent types can enforce the red-black invariant for balanced trees, but they cannot describe tree shapes.

Okasaki has used nested datatypes and rank-2 polymorphism to enforce data-structure shapes, such as the fact that a matrix is square [19]. We have not investigated his approach thoroughly, but it seems to suffice for "coordinated" examples over finite domains (such as tag bits), but not for infinite domains (such as closures' environment records).

Separation logic [20] can often express more sophisticated data invariants than type systems, but it appears no better equipped to abstract over an unbounded number of coordinated elements. Adapting our approach to a program logic could prove interesting.

Many type systems abstract over type lists (for example, consider row variables [24]), but the key to our work is developing the necessary peel coercion.

10 Conclusions and Future Work

Surprisingly modest extensions to low-level, polymorphic type systems can support coordinated data structures. With type trees and enriched recursive types, we have circumvented the type-variable scoping issues. We have demonstrated our extension's synergy with singleton integers and conditional types. For example, we have encoded a red-black tree using a pair of trees that are guaranteed to have the same shape.

In the future, we hope to make our ideas more practical. A key technical step is support for coordinated arrays. Typical arrays have (1) first-class index expressions and (2) mutation. The first is easy to handle, but mutable coordinated data may prove more challenging. We would also like to adapt our ideas for use in a source language or a modeling language. Inferring peel coercions seems crucial.

References

[1] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *4th ACM Symposium on Principles of Programming Languages*, pages 163–173, New York, NY, 1994.

[2] Andrew Appel. Foundational proof-carrying code. In *16th IEEE Symposium on Logic in Computer Science*, pages 247–258, 2001.

[3] Juan Chen, Dinghao Wu, Andrew W. Appel, and Hai Fang. A provably sound TAL for back-end optimization. In *ACM Conference on Programming Language Design and Implementation*, pages 208–219, 2003.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2nd edition, 2001.

[5] Karl Crary. Toward a foundational typed assembly language. In *30th ACM Symposium on Principles of Programming Languages*, pages 198–212, 2003.

[6] Karl Crary and Stephanie Weirich. Flexible type analysis. In *4th ACM International Conference on Functional Programming*, pages 233–248, 1999.

[7] Cormac Flanagan and Martín Abadi. Types for safe locking. In *8th European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108. Springer-Verlag, 1999.

[8] Dan Grossman. Existential types for imperative languages. In *11th European Symposium on Programming*, volume 2305 of *Lecture Notes in Computer Science*, pages 21–35. Springer-Verlag, 2002.

[9] Dan Grossman. Type-safe multithreading in Cyclone. In *ACM International Workshop on Types in Language Design and Implementation*, pages 13–25, 2003.

[10] Dan Grossman and Greg Morrisett. Scalable certification for typed assembly language. In *Workshop on Types in Compilation*, volume 2071 of *Lecture Notes in Computer Science*, pages 117–145. Springer-Verlag, 2000.

[11] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *ACM Conference on Programming Language Design and Implementation*, pages 282–293, 2002.

[12] Nadeem Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. In *17th IEEE Symposium on Logic in Computer Science*, pages 89–100, 2002.

[13] Nax Paul Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1–2):159–172, 1991.

[14] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *23rd ACM Symposium on Principles of Programming Languages*, pages 271–283, 1996.

[15] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *2nd ACM Workshop on Compiler Support for System Software*, pages 25–35, 1999. INRIA Technical Report 0288, 1999.

[16] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, 1999.

[17] Gapolan Nadathur. A notation for lambda terms II: Refinements and applications. Technical Report CS-1994-01, Duke University, 1994.

[18] George Necula. Proof-carrying code. In *24th ACM Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[19] Chris Okasaki. From fast exponentiation to square matrices: An adventure in types. In *4th ACM International Conference on Functional Programming*, pages 28–35, 1999.

[20] John Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[21] Michael F. Ringenburg and Dan Grossman. Type safety and erasure proofs for "A type system for coordinated data structures". Technical Report 2004-07-03, University of Washington, 2004.

[22] Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *3rd ACM International Conference on Functional Programming*, pages 313–323, 1998.

[23] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[24] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93:1–15, 1991.

[25] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[26] Hongwei Xi and Robert Harper. A dependently typed assembly language. In *6th ACM International Conference on Functional Programming*, pages 169–180, 2001.

[27] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *26th ACM Symposium on Principles of Programming Languages*, pages 214–227, 1999.

[28] http://www.cs.washington.edu/homes/miker/coord/.