Types for Describing Coordinated Data Structures

Michael F. Ringenburg[*]
miker@cs.washington.edu

Dan Grossman
djg@cs.washington.edu

Dept. of Computer Science & Engineering
University of Washington, Seattle, WA 98195

ABSTRACT

Coordinated data structures are sets of (perhaps unbounded) data structures where the nodes of each structure may share abstract types with the corresponding nodes of the other structures. For example, consider a list of arguments, and a separate list of functions, where the n-th function of the second list should be applied only to the n-th argument of the first list. We can guarantee that this invariant is obeyed by coordinating the two lists, such that the type of the n-th argument is existentially quantified and identical to the argument type of the n-th function. In this paper, we describe a minimal set of features sufficient for a type system to support coordinated data structures. We also demonstrate that two known type systems (Crary and Weirich's LX [6] and Xi, Chen and Chen's guarded recursive datatypes [24]) have these features, even though the systems were developed for other purposes. We illustrate the power of coordinated data structures as a programming idiom with three examples: (1) a list of function closures stored as a list of environments and a separate list of code pointers, (2) a "tagless" list, and (3) a red-black tree where the values and colors are stored in separate trees that are guaranteed to have the same shape.

Categories and Subject Descriptors: D.3.3 [Language Constructs and Features]: Data types and structures

General Terms: Languages

Keywords: Coordinated Data Structures

1. INTRODUCTION

This paper describes the features that polymorphic typed λ-calculi need in order to 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 n-th environment record corresponds to the n-th function). This section motivates why such invariants are important, explores why the scope of type variables makes the problem appear daunting, and previews the rest of the paper.

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, many low-level typed λ-calculi suffer from an unfortunate restriction resulting from the scope of type variables. For example, consider encoding closures of functions that have type $\mathsf{int} \rightarrow \mathsf{int}$ in a typed functional language such as Haskell or ML. A simple encoding is $\exists\alpha.\, \alpha \times ((\alpha \times \mathsf{int}) \rightarrow \mathsf{int})$. 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` [12]. (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 $\mathsf{int} \rightarrow \mathsf{int}$ in the source language have the same type after compilation (even if their environments have different types), which allows functions

[*] Supported in part by an ARCS fellowship sponsored by the Washington Research Foundation.

TLDI'05, January 10, 2005, Long Beach, California, USA.
Copyright 2005 ACM 1-58113-999-3/05/0001 ...$5.00.

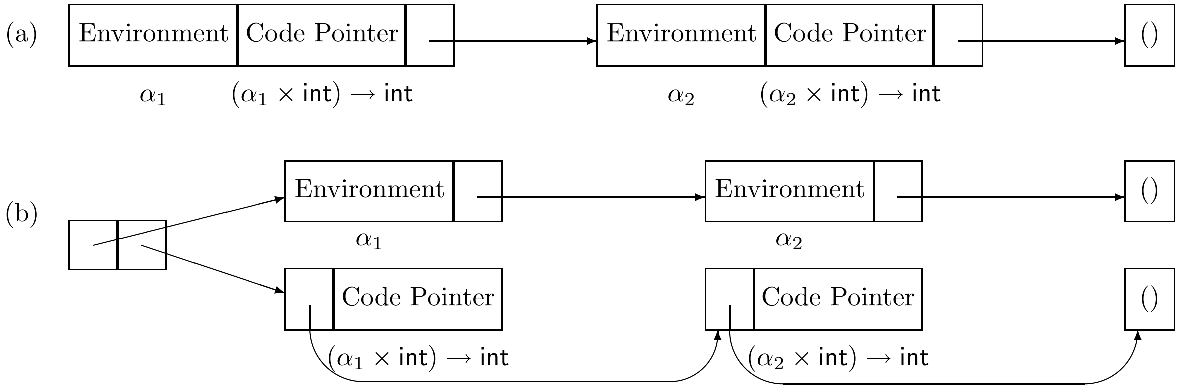

(a)
Environment
Code Pointer
Environment
Code Pointer
()
α_1
$(\alpha_1 \times \mathsf{int}) \to \mathsf{int}$
α_2
$(\alpha_2 \times \mathsf{int}) \to \mathsf{int}$
(b)
Environment
Environment
()
α_1
α_2
Code Pointer
Code Pointer
()
$(\alpha_1 \times \mathsf{int}) \to \mathsf{int}$
$(\alpha_2 \times \mathsf{int}) \to \mathsf{int}$

Figure 1: A function-closure list as (a) a conventional list, and (b) a coordinated pair of lists (using different list representations).

to be first-class. For example, a list of such functions could have type $\mu\beta.\mathsf{unit}+((\exists\alpha.\ \alpha \times ((\alpha \times \mathsf{int}) \to \mathsf{int})) \times \beta)$ (or another list encoding), where $\alpha+\beta$ represents a sum type with variants α and β. 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 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 do not seem 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 problem is fairly well-known, but to our knowledge it has never been investigated directly. Perhaps it was feared that supporting coordinated data structures would cause undesirable complications for what are already sophisticated languages. We argue, however, that such fear is unwarranted: With just a few additions to the polymorphic λ-calculus, we can support coordinated data structures while retaining type safety and syntax-directed type-checking.

Moreover, the additions—which by themselves are probably too special-purpose for direct use in a real type-checker—give us a direct way to judge other systems' power: If a system can encode our additions, it can support coordinated data structures. We know of two such systems: Crary and Weirich's LX [6] (originally designed for type analysis) and Xi, Chen, and Chen's guarded recursive datatypes [24] (originally designed for object encodings and staged computation). By using our minimal system to show that these systems are powerful enough to encode coordinated data structures, we provide additional incentive for their adoption.

1.3 Outline

The rest of this paper describes a simple type-theoretic way to allow coordinated data structure invariants in a typed λ-calculus, shows how we can use some existing languages to encode these invariants, and demonstrates their power through examples. Specifically, we:

- Explain how we use can use type lists, an enriched form of recursive type, and a new "peel" coercion to circumvent the type-variable scoping issues (Section 2).
- Build progressively more complex languages, starting with a language for coordinated lists (Section 3), extending it with singleton integers (Section 5), and finally supporting more general recursive coordinated types (Section 6). We illustrate each language with an extended example.
- Show how Crary and Weirich's LX [6] and Xi, Chen and Chen's guarded recursive datatypes [24] can encode coordinated data structures (Section 4).
- Discuss our prototype implementation (Section 7), related work (Section 8), and future work (Section 9).
- Establish type safety and type erasure for the simple coordinated list language (Appendix).

2. THE TRICK

We now describe a minimal and sufficient set of extensions to provide support for coordinated data structures in a polymorphic typed λ-calculus.

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 $\beta \rightarrow \mathsf{int}$ 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:

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

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]

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

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

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

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

$$\begin{array}{l} \exists\beta'{:}\mathrm{L}.(\ \ (\mu(\beta' \leftarrow \beta)\alpha.\mathsf{unit} + \beta \times \alpha) \\ \qquad\quad \times(\mu(\beta' \leftarrow \beta)\alpha.\mathsf{unit} + \alpha \times ((\beta \times \mathsf{int}) \rightarrow \mathsf{int}))) \end{array}$$

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:[2]

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

This rule allows two coordinated data structures; in practice peel should allow an n-tuple. As Section 3.2 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 particular coercion called peel.

3. LANGUAGE FOR COORDINATED LISTS

In this section, we present a simple language based on the extensions in Section 2. 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.

We emphasize that this language is powerful enough to encode only coordinated data structures where each node has at most one recursive child[3] (e.g., lists). In Section 6, we generalize the language to support coordinated data structures with multiple children (e.g., trees).

3.1 Syntax

Figure 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 $\tau \times \tau$ 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 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

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

[2] The rule as stated here requires e_1 to be a pair. In a low-level setting, constructing a pair requires an extra allocation. We could replace our peel with $\texttt{peel}\ e_1, \ldots, e_n\ \texttt{as}\ \alpha_{hd}, \alpha_{tl}, x_1, \ldots, x_n\ \texttt{in}\ e_{n+1}$, but the rule as stated here has the advantage of allowing a first class argument to peel.

[3] Technically, multiple recursive children can be supported, but only if the coordinated types are identical in every child. See Section 6.

$$\begin{array}{llcl}
\textsc{variables} & x & \in & \mathrm{Var} \\
\textsc{type variables} & \alpha,\beta & \in & \mathrm{Tyvar} \\
\textsc{kinds} & \kappa & ::= & \mathrm{T} \mid \mathrm{L} \\
\textsc{types} & \sigma,\tau & ::= & \mathsf{unit} \mid \alpha \mid \tau\times\tau \mid \tau+\tau \mid \tau\to\tau \mid \forall\alpha{:}\kappa.\tau \mid \exists\alpha{:}\kappa.\tau \mid \mu(\sigma\leftarrow\beta)\alpha.\tau \mid \tau^{*} \mid \tau{::}\sigma \\
\textsc{expressions} & e & ::= & () \mid x \mid (e,e) \mid \pi_i\ e \mid \mathsf{in}_i\ e \mid \mathsf{case}\ e\ \mathsf{of}\ x.e\ x.e \mid \lambda x{:}\tau.\ e \mid e\ e \mid \mathsf{fix}\ e \mid \Lambda\alpha{:}\kappa.\ e \mid e\ [\tau] \\
 & & \mid & \mathsf{pack}\ \tau,e\ \mathsf{as}\ \tau \mid \mathsf{unpack}\ e\ \mathsf{as}\ \alpha,x\ \mathsf{in}\ e \mid \mathsf{roll}\ e\ \mathsf{as}\ \tau \mid \mathsf{unroll}\ e \mid \mathsf{peel}\ e\ \mathsf{as}\ \alpha,\alpha,x\ \mathsf{in}\ e \\
\textsc{values} & v & ::= & () \mid (v,v) \mid \mathsf{in}_i\ v \mid \lambda x{:}\tau.\ e \mid \Lambda\alpha{:}\kappa.\ e \mid \mathsf{pack}\ \tau,v\ \mathsf{as}\ \tau \mid \mathsf{roll}\ v\ \mathsf{as}\ \tau \\
\textsc{type contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \\
\textsc{kind contexts} & \Delta & ::= & \cdot \mid \Delta, \alpha{:}\kappa
\end{array}$$

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

(see, for instance, [23]). 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 well-typed peel coercion never fails (gets stuck) at run-time. As the example in Section 3.4 shows, we can use unit* (or another *infinite*-list type) for *finite* terms like the empty-list.

3.3 Typing Rules

Typing judgments have the form $\Delta;\Gamma \vdash_{t} e : \tau$, 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

$$\begin{array}{c}\mathcal{P}_1\\\hline\mathcal{P}_2\\\mathcal{P}_3\end{array} \quad\text{as shorthand for}\quad \frac{\mathcal{P}_1}{\mathcal{P}_2} \quad\text{and}\quad \frac{\mathcal{P}_1}{\mathcal{P}_3}\ .$$

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, 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[4] and the code pointers[5]:

```
let t1  = ∃β′:L.(
     (μ(β′ ← β)α. unit + (β × α))
   ×(μ(β′ ← β)α. unit + (α × ((β × int) → int))))
```

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 $\alpha_1$,$\alpha_2$,x2 in
   case unroll($\pi_1$ x2) of
   x3. 0 (*1st list too short*)
   x3. case unroll($\pi_2$ x2) of
     x4. 0 (*2nd list too short*)
     x4. if n==1(*apply closure or recur*)
         then ($\pi_2$ x4) (($\pi_1$ x3), 0)
         else f (($\pi_2$ x3), ($\pi_1$ 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 (in$_2$ ($\pi_1$ c2, $\pi_1$ x2)) as
     μ(α′::β′ ← β)α. unit + (β × α)),
    (roll (in$_2$ ($\pi_2$ x2, $\pi_2$ c2)) as
     μ(α′::β′ ← β)α. unit + (α × ((β × int) → int))))
    as t1
```

One problem remains: How do we make the pair of empty lists for when there are no closures? For the recursive types, any σ suffices, so we can use unit*. Infinite lists of types ensure the peel coercion in apply_nth never gets stuck.

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

[5] We place the next-node pointer at the front of each node.

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

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

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

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

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

$$\boxed{\Delta \vdash_k \tau : \kappa}$$

KBASE

$$\frac{}{\begin{array}{c}\Delta \vdash_k \mathsf{unit} : \mathrm{T} \\ \Delta \vdash_k \alpha : \Delta(\alpha)\end{array}}$$

KPAIR

$$\frac{\Delta \vdash_k \tau_1 : \mathrm{T} \qquad \Delta \vdash_k \tau_2 : \mathrm{T}}{\begin{array}{c}\Delta \vdash_k \tau_1 \times \tau_2 : \mathrm{T} \\ \Delta \vdash_k \tau_1 + \tau_2 : \mathrm{T} \\ \Delta \vdash_k \tau_1 \to \tau_2 : \mathrm{T}\end{array}}$$

KQUANT

$$\frac{\Delta, \alpha{:}\kappa \vdash_k \tau : \mathrm{T}}{\begin{array}{c}\Delta \vdash_k \forall\alpha{:}\kappa.\tau : \mathrm{T} \\ \Delta \vdash_k \exists\alpha{:}\kappa.\tau : \mathrm{T}\end{array}}$$

KMU

$$\frac{\Delta \vdash_k \sigma : \mathrm{L} \qquad \Delta, \alpha{:}\mathrm{T}, \beta{:}\mathrm{T} \vdash_k \tau : \mathrm{T}}{\Delta \vdash_k \mu(\sigma \leftarrow \beta)\alpha.\tau : \mathrm{T}}$$

KSTAR

$$\frac{\Delta \vdash_k \tau : \mathrm{T}}{\Delta \vdash_k \tau^* : \mathrm{L}}$$

KCONS

$$\frac{\Delta \vdash_k \tau : \mathrm{T} \qquad \Delta \vdash_k \sigma : \mathrm{L}}{\Delta \vdash_k \tau{::}\sigma : \mathrm{L}}$$

$$\boxed{\Delta; \Gamma \vdash_t e : \tau}$$

BASE

$$\frac{}{\begin{array}{c}\Delta; \Gamma \vdash_t () : \mathsf{unit} \\ \Delta; \Gamma \vdash_t x : \Gamma(x)\end{array}}$$

INJECT

$$\frac{\Delta; \Gamma \vdash_t e : \tau \qquad \Delta \vdash_k \tau' : \mathrm{T}}{\begin{array}{c}\Delta; \Gamma \vdash_t \mathsf{in}_1\ e : \tau + \tau' \\ \Delta; \Gamma \vdash_t \mathsf{in}_2\ e : \tau' + \tau\end{array}}$$

PAIR

$$\frac{\Delta; \Gamma \vdash_t e_1 : \tau_1 \qquad \Delta; \Gamma \vdash_t e_2 : \tau_2}{\Delta; \Gamma \vdash_t (e_1, e_2) : \tau_1 \times \tau_2}$$

FUN

$$\frac{\Delta; \Gamma, x{:}\tau \vdash_t e : \tau' \qquad \Delta \vdash_k \tau : \mathrm{T}}{\Delta; \Gamma \vdash_t \lambda x{:}\tau.\ e : \tau \to \tau'}$$

CASE

$$\frac{\Delta; \Gamma \vdash_t e : \tau_1 + \tau_2 \qquad \Delta; \Gamma, x{:}\tau_1 \vdash_t e_1 : \tau \qquad \Delta; \Gamma, x{:}\tau_2 \vdash_t e_2 : \tau}{\Delta; \Gamma \vdash_t \mathsf{case}\ e\ \mathsf{of}\ x.e_1\ x.e_2 : \tau}$$

PROJ

$$\frac{\Delta; \Gamma \vdash_t e : \tau_1 \times \tau_2}{\begin{array}{c}\Delta; \Gamma \vdash_t \pi_1\ e : \tau_1 \\ \Delta; \Gamma \vdash_t \pi_2\ e : \tau_2\end{array}}$$

APP

$$\frac{\Delta; \Gamma \vdash_t e_1 : \tau' \to \tau \qquad \Delta; \Gamma \vdash_t e_2 : \tau'}{\Delta; \Gamma \vdash_t e_1\ e_2 : \tau}$$

FIX

$$\frac{\Delta; \Gamma \vdash_t e : \tau \to \tau}{\Delta; \Gamma \vdash_t \mathsf{fix}\ e : \tau}$$

TFUN

$$\frac{\Delta, \alpha{:}\kappa; \Gamma \vdash_t e : \tau}{\Delta; \Gamma \vdash_t \Lambda\alpha{:}\kappa.\ e : \forall\alpha{:}\kappa.\tau}$$

TAPP

$$\frac{\Delta; \Gamma \vdash_t e_1 : \forall\alpha{:}\kappa.\tau' \qquad \Delta \vdash_k \tau : \kappa}{\Delta; \Gamma \vdash_t e\ [\tau] : \tau'[\tau/\alpha]}$$

PACK

$$\frac{\Delta; \Gamma \vdash_t e : \tau'[\tau/\alpha] \qquad \Delta \vdash_k \tau : \kappa \qquad \Delta \vdash_k \exists\alpha{:}\kappa.\tau' : \mathrm{T}}{\Delta; \Gamma \vdash_t \mathsf{pack}\ \tau, e\ \mathsf{as}\ \exists\alpha{:}\kappa.\tau' : \exists\alpha{:}\kappa.\tau'}$$

UNPACK

$$\frac{\Delta; \Gamma \vdash_t e_1 : \exists\alpha{:}\kappa.\tau' \qquad \Delta, \alpha{:}\kappa; \Gamma, x{:}\tau' \vdash_t e_2 : \tau \qquad \Delta \vdash_k \tau : \mathrm{T}}{\Delta; \Gamma \vdash_t \mathsf{unpack}\ e_1\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ e_2 : \tau}$$

ROLL

$$\frac{\Delta; \Gamma \vdash_t e : \tau[\tau'/\beta][\mu(\sigma \leftarrow \beta)\alpha.\tau/\alpha] \qquad \Delta \vdash_k \mu(\tau'{::}\sigma \leftarrow \beta)\alpha.\tau : \mathrm{T}}{\Delta; \Gamma \vdash_t \mathsf{roll}\ e\ \mathsf{as}\ \mu(\tau'{::}\sigma \leftarrow \beta)\alpha.\tau : \mu(\tau'{::}\sigma \leftarrow \beta)\alpha.\tau}$$

UNROLL

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

PEEL

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

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

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

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

4. EXISTING LANGUAGES FOR COORDINATED DATA STRUCTURES

Now that we have identified a modest set of extensions that support coordinated data structures, the natural next step is to investigate whether or not any existing systems can encode these extensions and thus support coordinated data structures. In this section, we identify two such systems—Crary and Weirich's LX [6] (Section 4.1) and Xi, Chen and Chen's guarded recursive datatypes [24] (Section 4.2).

4.1 LX

Crary and Weirich's formal language LX [6] was originally designed for flexible run-time type analysis. In essence, it provides a rich but strongly-normalizing programming language at the type level. In this section, we show that LX can encode the extensions described in Section 3, and can thus support coordinated data structures.

We can encode our type lists with LX's inductive kinds:

$$\mathrm{L} = \mu j.\mathsf{Type} + (\mathsf{Type} \times j)$$

The left side of the above sum type represents lists of the form τ^* and the right side represents lists of the from $\tau{::}\sigma$. We use $\mathsf{fold}_L(\mathsf{inj}_1^{\mathsf{Type}+(\mathsf{Type}\times j)}\tau)$ to create lists of the form τ^*, and $\mathsf{fold}_L(\mathsf{inj}_2^{\mathsf{Type}+(\mathsf{Type}\times j)}(\tau,\sigma))$ to create lists of the form $\tau{::}\sigma$. In an ML-style syntax, we could instead write the list kind as:

```
kind L = Star of Type      (* τ* *)
       | Cons of Type * L  (* τ::σ *)
```

The type-level primitive recursion of LX lets us deconstruct our inductive list kind and define functions that return the head and tail of a list:

```
head = pr(j,α : Type + (Type × j), φ : j → Type .
         case α of
           inj₁ β  ⇒  β
           inj₂ (β,γ)  ⇒  β
tail = pr(j,α : Type + (Type × j), φ : j → L .
         case α of
           inj₁ β  ⇒ fold(inj₁ β)
           inj₂ (β,γ)  ⇒  γ
```

Both functions simply check whether the list is of the form τ^* or the form $\tau{::}\sigma$, and return the appropriate result. In an ML-style syntax, we could instead write these functions as:

```
head(Star(t)) = t        (* head(τ*) = τ *)
head(Cons(t,s)) = t      (* head(τ::σ) = τ *)
tail(Star(t)) = Star(t)  (* tail(τ*) = τ* *)
tail(Cons(t,s)) = s      (* tail(τ::σ) = σ *)
```

We can define our peel coercion as:

```
peel(α) = Cons(head(α),tail(α))
```

Finally, the enriched recursive types $\mu(\sigma \leftarrow \beta)\alpha.\tau$ from section 3 become $\mathsf{rec}_{\mathsf{L}}(c_1,\sigma)$, where

$$c_1 = \lambda\phi : L \rightarrow \mathsf{Type}.\lambda\beta : L.\tau[\mathsf{head}(\beta)/\beta][(\phi(\mathsf{tail}(\beta)))/\alpha]\ .$$

When unrolled, the ϕs in τ are replaced with a function which takes an argument π and returns a new recursive type $\mathsf{rec}_{\mathsf{L}}(c_1,\pi)$ parameterized by π. The βs in τ are replaced with the original parameter σ. Thus, this type unrolls and reduces to

$$\tau[\mathsf{head}(\sigma)/\beta][\mathsf{rec}_{\mathsf{L}}(c_1,\mathsf{tail}(\sigma))/\alpha]\ ,$$

which is equivalent to the unrolling of the enriched recursive types presented in section 3.

4.2 Guarded Recursive Datatypes

Xi, Chen, and Chen's guarded recursive datatypes [24] were initially developed for run-time type passing. In this section, we show that they are also sufficiently powerful to encode our extensions, and thus to represent coordinated data structures.

We illustrate the encoding of our extensions through the example of a pair of coordinated lists. We use an abstract type constructor tylst to represent our type lists:

$$\mathsf{type\ tylst}\ \alpha_{hd}\ \alpha_{tl}$$

The parameters α_{hd} and α_{tl} represent the list head and tail, respectively. We use two algebraic datatypes parameterized by tylst to represent the types of our two coordinated lists. Our first algebraic type corresponds to a list of type $\mu(\sigma \leftarrow \beta)\alpha.\mathsf{unit} + \beta \times \alpha$ in the language of Section 3:

```
typecons (type) listone =
  {α}        . (α) Nil
| {αtl,αhd} . (tylst αhd αtl) Cons of
                         αhd * (αtl listone)
```

The syntax $\{\alpha_{tl},\alpha_{hd}\}$ can be read "for all α_{tl} and α_{hd}", and the syntax α_{tl} $\mathsf{listone}$ represents a $\mathsf{listone}$ parameterized with type α_{tl}. The second algebraic type

```
typecons (type) listtwo =
  {α}        . (α) Nil
| {αtl,αhd} . (tylst αhd αtl) Cons of
                          (αhd -> int) * (αtl listtwo)
```

corresponds to

$$\mu(\sigma \leftarrow \beta)\alpha.\mathsf{unit} + ((\beta \rightarrow \mathsf{int}) \times \alpha)$$

in Section 3. In general, if we have an enriched recursive type of the form $\mu(\sigma \leftarrow \beta)\alpha.\tau_1 + \tau_2 + \ldots$ where τ_1 is a leaf type (i.e, contains no recursive children α), we can encode it with the guarded recursive type

```
typecons (type) foo =
  {α}    . (α) Foo1 of τ1[α/β]
| {α,β} . (tylst β α) Foo2 of τ2[α foo/α]
| ...
```

We coordinate a list of type $\mathsf{listone}$ with a list of type $\mathsf{listtwo}$ by creating a third type that represents pairs of lists with equivalent type lists:

```
type lst1_lst2_pair =
  {β′} . Pair of (β′ listone) * (β′ listtwo)
```

$$
\begin{array}{lcl}
\kappa & ::= & \ldots \mid \mathrm{I} \mid \mathrm{L_I} \\
\tau & ::= & \ldots \mid i \mid \mathrm{S}(\tau) \mid \mathsf{if}\ \tau\ \mathsf{then}\ \tau\ \mathsf{else}\ \tau \\
e & ::= & \ldots \mid i \mid \mathsf{tosum}\ e, \tau \mid \mathsf{match}\ e\ x.e\ x.e \\
v & ::= & \ldots \mid i \mid \mathsf{tosum}\ v, \tau
\end{array}
$$

$$
\begin{array}{l}
E ::= \ldots \mid \mathsf{tosum}\ E, \tau \mid \mathsf{match}\ E\ x.e\ x.e \\
\mathsf{match}\ (i, (\mathsf{tosum}\ v, E))\ x.e_1\ x.e_2 \xrightarrow{\mathrm{r}} e_1[v/x] \quad \text{if } i \neq 0 \\
\mathsf{match}\ (i, (\mathsf{tosum}\ v, E))\ x.e_1\ x.e_2 \xrightarrow{\mathrm{r}} e_2[v/x] \quad \text{if } i = 0
\end{array}
$$

$$
\frac{i \in \mathcal{Z}}{\Delta \vdash_{\mathrm{k}} i : \mathrm{I}}
\qquad
\frac{\Delta \vdash_{\mathrm{k}} \tau : \mathrm{I}}{\Delta \vdash_{\mathrm{k}} \mathrm{S}(\tau) : \mathrm{T}}
\qquad
\frac{i \in \mathcal{Z}}{\Delta; \Gamma \vdash_{\mathrm{t}} i : \mathrm{S}(i)}
\qquad
\frac{\Delta \vdash_{\mathrm{k}} \tau_1 : \mathrm{I} \qquad \Delta \vdash_{\mathrm{k}} \tau_2 : \mathrm{T} \qquad \Delta \vdash_{\mathrm{k}} \tau_3 : \mathrm{T}}{\Delta \vdash_{\mathrm{k}} \mathsf{if}\ \tau_1\ \mathsf{then}\ \tau_2\ \mathsf{else}\ \tau_3 : \mathrm{T}}
$$

$$
\frac{\Delta \vdash_{\mathrm{k}} \sigma : \mathrm{L_I} \qquad \Delta, \alpha{:}\mathrm{T}, \beta{:}\mathrm{I} \vdash_{\mathrm{k}} \tau : \mathrm{T}}{\Delta \vdash_{\mathrm{k}} \mu(\sigma \leftarrow \beta)\alpha.\tau : \mathrm{T}}
\qquad
\frac{\Delta \vdash_{\mathrm{k}} \tau : \mathrm{I}}{\Delta \vdash_{\mathrm{k}} \tau^* : \mathrm{L_I}}
\qquad
\frac{\Delta \vdash_{\mathrm{k}} \tau : \mathrm{I} \qquad \Delta \vdash_{\mathrm{k}} \sigma : \mathrm{L_I}}{\Delta \vdash_{\mathrm{k}} \tau{::}\sigma : \mathrm{L_I}}
$$

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

$$
\frac{\begin{array}{c}\Delta; \Gamma, x{:}\tau_2 \vdash_{\mathrm{t}} e_2 : \tau_4 \qquad \Delta; \Gamma, x{:}\tau_3 \vdash_{\mathrm{t}} e_3 : \tau_4 \\ \Delta; \Gamma \vdash_{\mathrm{t}} e_1 : \mathrm{S}(\tau_1) \times \mathsf{if}\ \tau_1\ \mathsf{then}\ \tau_2\ \mathsf{else}\ \tau_3\end{array}}{\Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{match}\ e_1\ x.e_2\ x.e_3 : \tau_4}
\qquad
\frac{\Delta; \Gamma \vdash_{\mathrm{t}} e : \tau_1 \qquad \Delta \vdash_{\mathrm{k}} \tau_2 : \mathrm{T} \qquad i \neq 0}{\begin{array}{c}\Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{tosum}\ e, \mathsf{if}\ i\ \mathsf{then}\ \tau_1\ \mathsf{else}\ \tau_2 : \mathsf{if}\ i\ \mathsf{then}\ \tau_1\ \mathsf{else}\ \tau_2 \\ \Delta; \Gamma \vdash_{\mathrm{t}} \mathsf{tosum}\ e, \mathsf{if}\ 0\ \mathsf{then}\ \tau_2\ \mathsf{else}\ \tau_1 : \mathsf{if}\ 0\ \mathsf{then}\ \tau_2\ \mathsf{else}\ \tau_1\end{array}}
$$

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

This type is roughly equivalent to:

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

In order to use a coordinated pair of lists in the language of Section 3, we must first transform their type lists into a form usable by the unroll coercion. We must also ensure that the two lists continue to be parameterized by identical type lists—otherwise the two lists will no longer be coordinated. We use the peel coercion to accomplish the needed equality-preserving transformation.

However, in the guarded recursive datatype encoding, peel is no longer necessary. Pattern matching unpacks and unrolls the lists, and the type equalities in the guarded type-variable contexts ensure that the equivalence of the two type lists is not forgotten. For instance, we could write a function that applies the first element of a listtwo to the first element of a listone as follows:

```
let apply1 (l1: α listone) (l2: α listtwo) : int =
match l1 with
  Nil -> 0
| Cons (x,_) -> (* x:β₁, α = tylst β₁ α₁ *)
  match l2 with
    Nil -> 0
  | Cons (y,_) -> (* y:β₂ → int, α = tylst β₂ α₂ *)
      (y x)
```

The first Cons pattern introduces the constraint that `l1` is parameterized with some `tylst` $\beta_1\ \alpha_1 = \alpha$ and the second shows that `l2` is parameterized with a `tylst` $\beta_2\ \alpha_2 = \alpha$. The theory of type equality then concludes both type lists are the same, and thus `(y x)` is type safe. The τ^* lists are present in Section 3 only to ensure peel is defined for all list-types since peel must precede unroll. A pattern-match that simultaneously unrolls a recursive type and introduces type equalities fills both roles.

4.3 Discussion

We have now seen three systems that support coordinated data structures. The minimal language in Section 3 is, by design, a direct approach with syntax-directed typechecking. LX and guarded recursive datatypes have rich notions of type equality: The former uses normalization of a powerful type-level language and the latter uses explicit equalities introduced by quantified types. In both cases, we have established the nonobvious claim that the system supports coordinated data structures by showing that it can encode the direct approach's type lists, recursive types, and peel coercions. For LX, this amounted to programming with primitive recursion. For guarded recursive datatypes, we showed how pattern-matching exposes the same type equality that the peel coercion realizes via substitution. Using pattern-matching to combine unroll, case, unpack, and the introduction of a type equality leads to a particularly parsimonious solution.

5. SINGLETON-INTEGER SYNERGY

In this section, we discuss the synergy between coordinated data structures and singleton-integer types. We show how the combination of the two lets us create a pair of lists where only one list has tags.

Standard implementations of typed recursive data structures require run-time *tags* indicating whether or not a node has recursive children. For instance, in a list with type $\mu\alpha.\mathsf{unit} + \mathsf{int} \times \alpha$, each node carries a tag indicating whether it has type unit or type $\mathsf{int} \times \alpha$. These tags seem necessary to discriminate between node types.

We can partially eliminate tags, though, by combining coordinated data structures with *singleton integers* and *conditional types* (see, e.g., [1]). Specifically, we can create a pair of coordinated lists[6] 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

[6] We can do the same thing for general recursive data structures using the extensions described in Section 6.

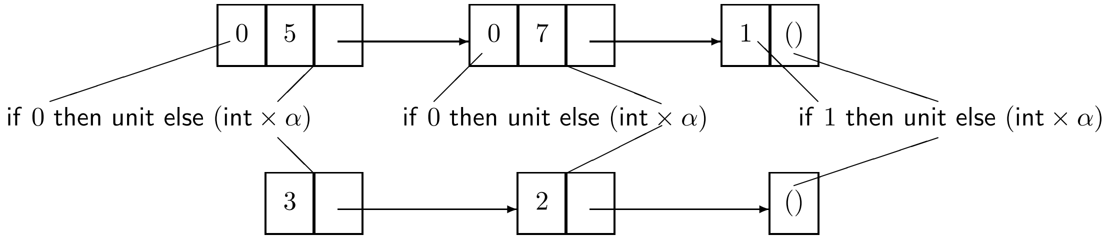

0
5
0
7
1
()
if 0 then unit else (int × α)
if 0 then unit else (int × α)
if 1 then unit else (int × α)
3
2
()

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.

$$\text{let } \tau^m = \tau_1, \tau_2, \ldots, \tau_m \qquad \begin{array}{rcl} \kappa & ::= & \ldots \mid \mathrm{L}^{(m,n)} \mid \mathrm{L_I}^{(m,n)} \\ \sigma, \tau & ::= & \ldots \mid \mu(\sigma \leftarrow (\beta^m))(\alpha^n).\tau \mid (\tau^m)^{*n} \mid (\tau^m)::^{m,n}(\sigma^n) \\ e & ::= & \ldots \mid \mathsf{peel}\ e\ \mathsf{as}\ \alpha^m, \beta^n, x\ \mathsf{in}\ e \\ E & ::= & \ldots \mid \mathsf{peel}\ E\ \mathsf{as}\ \alpha^m, \beta^n, x\ \mathsf{in}\ e \end{array}$$

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

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

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

tag of the corresponding node in the tagged list. Neither singleton integers nor conditional types are new—it is their combination with coordinated data structures that enables this encoding.

To demonstrate how this encoding works, we extend our language to include singleton integers. Figure 5 contains the necessary changes. We add a new integer kind (I), a new list kind for lists of integer types ($\mathrm{L_I}$), 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. We also add new typing rules for integers and lists of integer types.

Figure 5 also contains 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 deconstructing 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, [13] and [25]).

These additions let us create a coordinated pair consisting of a tagged and a tagless list (see Figure 6). We give the pair the type:

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

The first list is tagged (the $\mathrm{S}(\beta)$), 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). 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 that has a type of the form $\mathrm{S}(i) \times \mathsf{if}\ i\ \mathsf{then}\ \tau_1\ \mathsf{else}\ \tau_2$. 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 correct type: the tag of the corresponding element of the tagged list.

We have written `zip` and `unzip` functions for tagged-tagless pairs of lists (see our website [27]). The `zip` function converts a tagged-tagless pair into a single list of pairs, and `unzip` is its inverse.

6. FULL LANGUAGE FOR COORDINATED DATA STRUCTURES

In this section, we generalize our language from Sections 3 and 5 to support recursive data structures with multiple children (e.g., trees). We use this extension to show how coordinated data structures can 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. The 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.

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

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

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

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

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

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

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

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

6.1 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

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

If we try to encode this tree as a coordinated pair of trees we will end up with a type of the form

$$\begin{array}{l}\exists\sigma{:}\mathrm{L}.((\mu(\sigma \leftarrow \beta)\alpha.\textsf{unit} + (\beta \times (\alpha \times \alpha)))\times \\ \qquad (\mu(\sigma \leftarrow \beta)\alpha.\textsf{unit} + (\beta \times (\alpha \times \alpha))))\ .\end{array}$$

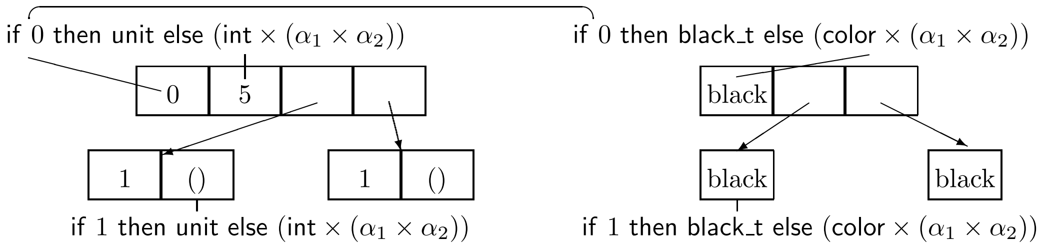
When we peel this pair, we get back a single α_{tl} list tail. When we unroll one of the trees, the same tail list will be used for *both* children. So each child will share types drawn from σ not only with the corresponding child in the *other* tree, but also with its sibling in the *same* tree. In 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 new kinds $\mathrm{L}^{(m,n)}$ and $\mathrm{L_I}^{(m,n)}$ for n-ary type trees with m coordinated types per node. For example, our red-black tree example will use type trees of kind $\mathrm{L_I}^{(1,2)}$. 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 $\mathrm{L}^{(m,n)}$ (or $\mathrm{L_I}^{(m,n)}$), 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 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 $\alpha_{tl,j}$.

Figure 8 presents the modified typing rules. The kinding rules for μ, *, and :: types 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.



if 0 then unit else (int × $(\alpha_1 \times \alpha_2)$)
0
5
1
()
1
()
if 1 then unit else (int × $(\alpha_1 \times \alpha_2)$)
if 0 then black_t else (color × $(\alpha_1 \times \alpha_2)$)
black
black
black
if 1 then black_t else (color × $(\alpha_1 \times \alpha_2)$)

Figure 9: A red-black tree encoded as a pair of coordinated trees.

LX and guarded recursive datatypes support this generalization by increasing the arity of type constructors appropriately, which is the essence of the generalization.

6.2 Split Red-Black Trees

In this section, we describe a coordinated data structure implementation of red-black trees. Our implementation uses coordinated data structures 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 5, 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 =

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

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.

We have implemented lookup and insert functions for our tree-pairs (see Section 7). The lookup procedure accesses only the value tree.

7. IMPLEMENTATION

We wrote an interpreter for our language in O'Caml and verified that the examples in the 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 [27].

Our red-black tree implementation includes an empty tree, a lookup function, and an insert function. The lookup function 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 both trees.

8. RELATED WORK

Typed assembly languages and proof-carrying code frameworks (e.g, [14, 15, 10, 2, 3, 5]) aim to provide expressive type languages so that compilers can choose natural and efficient data representations. Many have singleton types (which have uses such as enforcing lock-based mutual exclusion [7, 8] or region-based memory management [21, 9]), so supporting coordinated data structures (with any of the three systems we have demonstrated) would provide the synergy we have demonstrated.

Guarded recursive datatypes have generated significant recent attention [24, 18, 17, 11], suggesting that there may be sufficient interest to see their widespread adoption. For example, Pottier and Régis-Gianas' [18] parser generator uses guarded types to produce parsers that are both efficient and type-safe.

Xi's work on dependent types [26, 25] has expressiveness that overlaps with our type system, 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 [16]. 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 [19] 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 [22]). The key for coordinated data structures, however, is having a way to implement the peel coercion, whether explicitly or through pattern matching (as in Section 4.2).

9. CONCLUSIONS AND FUTURE WORK

Surprisingly modest extensions to low-level, polymorphic type systems can support coordinated data structures. We circumvented the type-variable scoping problems with type trees and enriched recursive types. We also demonstrated our extension's synergy with singleton integers and conditional types. For example, we encoded a red-black tree using a pair of trees that are guaranteed to have the same shape. We also showed that our system is a useful tool for identifying other systems that support coordinated data structures. In particular, we showed that Crary and Weirich's LX [6] and Xi, Chen and Chen's guarded recursive datatypes [24] are sufficiently powerful to encode coordinated data structures, although neither was designed for this purpose.

In the future, we hope to investigate systems 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 know of no existing systems with such support.

10. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers of earlier versions for improving our understanding of guarded recursive datatypes and providing an example similar to the one in Section 4.2, as well as improving our presentation of singleton integers. We would also like to thank Sorin Lerner, David Walker, and Matthew Fluet for helpful comments on a previous draft of this paper.

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

[9] 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.

[10] 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.

[11] Ralf Hinze. Fun with phantom types. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, pages 245–262. Palgrave Macmillan, 2003.

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

[13] 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.

[14] 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.

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

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

[17] Francois Pottier and Nadji Gauthier. Polymorphic typed defunctionalization. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL'04)*, pages 89–98, Venice, Italy, January 2004.

[18] Francois Pottier and Yann Régis-Gianas. Towards efficient, typed LR parsers. Submitted to the *Journal of Functional Programming*, September 2004.

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

[20] 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.

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

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

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

[24] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *30th ACM Symposium on Principles of Programming Languages*, pages 224–235, 2003.

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

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

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

APPENDIX

This appendix describes important metatheoretic results for the language presented in Section 3. We outline our type-safety proof and briefly describe our type-erasure theorem.

Type Safety

The accompanying technical report [20] contains detailed proofs of all lemmas. Here we consider only a few lemmas and their proofs' most interesting cases.

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 [23].

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 $e' =$

  $e_1[\tau'/\alpha_{hd}][\sigma'/\alpha_{tl}]$
  $\quad [(\mathsf{roll}\ v_1\ \mathsf{as}\ \mu(\tau'{::}\sigma' \leftarrow \beta)\alpha.\tau_1,$
  $\quad\quad \mathsf{roll}\ v_2\ \mathsf{as}\ \mu(\tau'{::}\sigma' \leftarrow \beta)\alpha.\tau_2)/x]$

  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)

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

  (3)

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

  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 $e \rightarrow e'$.*

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
  $\cdot;\cdot \vdash_{\mathrm{t}} e_1 : \mu(\sigma \leftarrow \beta)\alpha.\tau_1 \times \mu(\sigma \leftarrow \beta)\alpha.\tau_2$.
  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
  $(\mathsf{roll}\ v_1\ \mathsf{as}\ \mu(\sigma \leftarrow \beta)\alpha.\tau_1, \mathsf{roll}\ v_2\ \mathsf{as}\ \mu(\sigma \leftarrow \beta)\alpha.\tau_2)$.

  $$\begin{aligned} \text{Thus } e \xrightarrow{\mathrm{r}} e_2[\tau'/\alpha_{hd}][\sigma'/\alpha_{tl}] & \\ [(\mathsf{roll}\ v_1\ \mathsf{as}\ \mu(\tau'{::}\sigma' \leftarrow \beta)\alpha.\tau_1, & \\ \mathsf{roll}\ v_2\ \mathsf{as}\ \mu(\tau'{::}\sigma' \leftarrow \beta)\alpha.\tau_2)/x] & \end{aligned}$$

  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.

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 [20]. The rule for `peel` is typical for coercions:

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

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 $e \rightarrow^{\star} v$, then $\mathtt{erase}(e) \rightarrow^{\star} \mathtt{erase}(v)$ in the untyped language. (Also, e and $\mathtt{erase}(e)$ have the same termination behavior.)