An Open and Shut Typecase

Dimitrios Vytiniotis Geoffrey Washburn Stephanie Weirich
Department of Computer and Information Science
University of Pennsylvania
{ dimitriv,geoffw,sweirich }@cis.upenn.edu

ABSTRACT

Two different ways of defining ad-hoc polymorphic operations commonly occur in programming languages. With the first form polymorphic operations are defined inductively on the structure of types while with the second form polymorphic operations are defined for specific sets of types.

In intensional type analysis operations are defined by induction on the structure of types. Therefore no new cases are necessary for user-defined types, because these types are equivalent to their underlying structure. However, intensional type analysis is "closed" to extension, as the behavior of the operations cannot be differentiated for the new types, thus destroying the distinctions that these types are designed to express.

Haskell type classes on the other hand define polymorphic operations for sets of types. Operations defined by class instances are considered "open"—the programmer can add instances for new types without modifying existing code. However, the operations must be extended with specialized code for each new type, and it may be tedious or even impossible to add extensions that apply to a large universe of new types.

Both approaches have their benefits, so it is important to let programmers decide which is most appropriate for their needs. In this paper, we define a language that supports both forms of ad-hoc polymorphism, using the same basic constructs.

Categories and Subject Descriptors

D.3.3 [**PROGRAMMING LANGUAGES**]: Language Constructs and Features—*abstract data types, polymorphism, control structures*; F.3.3 [**LOGICS AND MEANINGS OF PROGRAMS**]: Software—*type structure, program and recursion schemes, functional constructs*; F.4.1 [**MATHEMATICAL LOGIC AND FORMAL LANGUAGES**]: Mathematical Logic—*Lambda calculus and related systems*

General Terms

Languages, Theory

Keywords

ad-hoc polymorphism, generativity, intensional type analysis, reflexivity

1. INTRODUCTION

With ad-hoc polymorphism the execution of programs depends on type information. We call operations that depend on type information *type-directed*. Ad-hoc polymorphism can be used to implement dynamic typing, dynamic loading and marshaling. It is also essential to the definition of generic versions of many basic operations such as equality and structural traversals. Therefore, ad-hoc polymorphism can significantly simplify programming with complicated data structures, eliminating the need for repetitive "boilerplate code".

There are two ways to think about how ad-hoc polymorphic operations may be defined in a programming language. The first way is to think that these operations are defined inductively over the *structure* of types. For example, with intensional type analysis [13] we may define polymorphic equality as follows:

```
eq (x::a) (y::a) =
   typecase a of
     Int    -> eqint x y
   | Bool   -> if x then y else not y
   | (b,c)  -> eq (fst x)(fst y) &&
               eq (snd x)(snd y)
   | [b]    -> all2 eq x y
   | (b->c) -> error "eq not defined for functions"
```

The `typecase` term determines the structure of the type and calls the equality function recursively on the subcomponents of the type. For type soundness, the analysis of types has to cover all types provided by the language. This is somewhat awkward as some cases may be nonsensical for a given operation. For example, the case for functions above has to be provided even though we do not wish to define equality for functions.

The second way to think about ad-hoc polymorphic operations is to think of them as operations defined for a specific *set* of types. For example, instead of defining polymorphic equality for all types, with Haskell type classes [34] we may specify a set of type names for which equality is defined. The type class `Eq` below declares that there is an operation called `eq`.

```
class Eq a where
  eq :: a -> a -> Bool
```

The behavior of the operation is specified by the instances of the type class. Each instance of the type class describes how eq behaves for the given type. For composite types, such as products, equality is defined in terms of equality for the components of the type.

```
instance Eq Int where
  eq x y = eqint x y
instance Eq Bool where
  eq x y = if x then y else not y
instance (Eq a, Eq b) => Eq (a,b) where
  eq (x1,y1) (x2,y2) = eq x1 x2 && eq y1 y2
```

Type classes are "open" to extension. Instances do not need to be defined all at once. The programmer can add new instances anywhere in the program without modifying existing code.

Ad-hoc polymorphism becomes more complicated in languages with user-defined types. Statically, user defined types can be considered *equal* or *isomorphic* to their underlying definitions. When they are considered equal to their underlying definitions they are usually called type abbreviations and the type checker does not make any distinctions between the user-defined type and its definition; the type and its definition are interchangeable in all contexts. On the other hand, when user defined types are considered isomorphic to their underlying definitions, the type checker is able to enforce the application-specific distinctions that these types are designed to express. The programmer can then convert between the type and its definition by using special coercion operations. For example, consider the following Haskell `newtype` [25] definition.

```
newtype Phone = P Int
```

This new type can distinguish phone numbers from other integers in a program, helping the programmers avoid confusion. The data constructor `P` is just a coercion between the types `Int` and `Phone` allowing the creation of terms of this type. Pattern matching in Haskell can be used to define the opposite coercion.

In intensional type analysis, operations defined by `typecase` are "closed" to extension because they are defined inductively on the structure of types. The programmer does not have a way to modify or extend the behavior of polymorphic operations to new types. In contrast, Haskell type classes distinguish between a new type and its definition. For example, even if `Int` is a member of the equality class, the type `Phone` is not automatically.

Making this distinction is desirable because it is more expressive: Sometimes operations may need to take advantage of such distinctions. For example, programmers may want a polymorphic serializer to have different behavior on integers and phone numbers. However, in some cases not making the distinction is also desirable because it saves a lot of work. For example, programmers may want structural equality defined for all types in their programs without having to declare instances for each new type they define.

The trade-offs between these two mechanisms are related to the *expression problem* [37] but at the level of types instead of terms. The expression problem occurs when we have datatypes defined by cases and functions over these datatypes. Suppose that we want to extend a datatype with a new case, but we do not want to modify or recompile our existing functions, nevertheless retaining type safety. At the level of types we would like to be able to add both new cases (i.e. user-defined types) and new functions (i.e. type-analyzing operations). With intensional type analysis adding new operations is easy but new user-defined types are not really treated as new. With type classes we can add meaningful new user-defined types but extending functions or adding new functions is difficult: we have to change the code in various places. Moreover if a new type is defined in a separate inaccessible module, then it is impossible for the programmer to extend an operation to that type. Instead he must rely on the definer of the type to add the instance; but there is no guarantee that the definer of the type will respect the invariants of the operation.

Neither intensional type analysis nor Haskell type classes are appropriate for all cases, so we would like to combine the characteristics of both using a single mechanism.

1.1 Combining both forms in one language

This paper unifies the characteristics of the two forms of ad-hoc polymorphism in a foundational language, called $\lambda_{\mathcal{L}}$. It allows developers to choose which characteristics they want to use from each system, instead of forcing them to make this decision a priori.

At the core, $\lambda_{\mathcal{L}}$ is a system for structural type analysis augmented with user-defined types, represented as labels. The structural type analysis operator **typecase** may include branches for user-defined types. Types containing labels for which there is no branch in an operation cannot be allowed as arguments, or evaluation will become stuck. Therefore, the $\lambda_{\mathcal{L}}$ type system tracks the labels used in types and compares them to the domain of a type analysis operation.

The type analysis provided by $\lambda_{\mathcal{L}}$ is *extensible* to new user-defined types represented by labels, generated dynamically during execution. For this purpose, we introduce first-class maps from labels to expressions. Intuitively, these maps are branches for **typecase** that may be passed as arguments to type-directed operations, extending them to handle the new labels.

To complement its other features, $\lambda_{\mathcal{L}}$ includes support for coercing the types of expressions so that new type labels that are mentioned in them may be replaced by their underlying definitions. This way values whose types mention some new types may be used with operations that do not include cases for these new types.

When viewed as a programming language, $\lambda_{\mathcal{L}}$ requires that programs be heavily annotated and written in a highly-stylized fashion. The next step in this research is the design of an easy-to-program-in source language that will expose the advanced features of $\lambda_{\mathcal{L}}$ and that will incorporate automated assistance for common idioms, such as inference of type arguments.

Technical material, as well as the implementation of an explicitly typed language based on a fully reflexive variant of $\lambda_{\mathcal{L}}$ can be found at:

`http://www.cis.upenn.edu/~dimitriv/itaname/`

1.2 Contributions of this work

This paper makes the following contributions:

- We define a language that allows the definition of both

$$
\begin{array}{rcll}
\multicolumn{4}{l}{\textit{Kinds}} \\
\kappa & ::= & \star \mid \kappa_1 \rightarrow \kappa_2 & \\
\multicolumn{4}{l}{\textit{Labels}} \\
l & ::= & \iota \mid \ell_i^{\kappa} & \textit{variables and constants} \\
\multicolumn{4}{l}{\textit{Label sets}} \\
\mathcal{L} & ::= & s & \textit{variables} \\
 & \mid & \varnothing \mid \mathcal{U} & \textit{empty and universe} \\
 & \mid & \{l\} \mid \mathcal{L}_1 \cup \mathcal{L}_2 & \textit{singleton and union} \\
\multicolumn{4}{l}{\textit{Types}} \\
\tau & ::= & \alpha \mid \lambda\alpha{:}\kappa.\tau \mid \tau_1\ \tau_2 & \lambda\textit{-calculus} \\
 & \mid & l & \textit{labels} \\
 & \mid & \forall\alpha{:}\kappa{\upharpoonright}\mathcal{L}.\tau & \textit{type of type-poly. terms} \\
 & \mid & \forall\iota{:}\mathrm{L}(\kappa).\tau & \textit{type of label-poly. terms} \\
 & \mid & \forall s{:}\mathrm{Ls}.\tau & \textit{type of set-poly. terms} \\
 & \mid & \mathcal{L}_1{\Rightarrow}\tau{\upharpoonright}\mathcal{L}_2 & \textit{type of typecase branches} \\
\multicolumn{4}{l}{\textit{Terms}} \\
e & ::= & x \mid \lambda x{:}\tau.e \mid e_1\ e_2 & \lambda\textit{-calculus} \\
 & \mid & i \mid \mathbf{fix}\ x{:}\tau.e & \textit{integers and recursion} \\
 & \mid & \mathbf{new}\ \iota{:}\kappa = \tau\ \mathbf{in}\ e & \textit{label creation} \\
 & \mid & \{\!\{e\}\!\}^{\pm}_{l=\tau} & \textit{first-order coercion} \\
 & \mid & \{\!\{e : \tau'\}\!\}^{\pm}_{l=\tau} & \textit{higher-order coercion} \\
 & \mid & \mathbf{typecase}\ \tau\ e & \textit{type analysis} \\
 & \mid & \varnothing \mid \{l \Rightarrow e\} \mid e_1 \bowtie e_2 & \textit{branches} \\
 & \mid & \Lambda\alpha{:}\kappa{\upharpoonright}\mathcal{L}.e \mid e[\tau] & \textit{type polymorphism} \\
 & \mid & \Lambda\iota{:}\mathrm{L}(\kappa).e \mid e[\hat{l}] & \textit{label polymorphism} \\
 & \mid & \Lambda s{:}\mathrm{Ls}.e \mid e[\mathcal{L}] & \textit{label set polymorphism}
\end{array}
$$

Figure 1: The core $\lambda_{\mathcal{L}}$ language

"open" and "closed" type-directed operations. Previous work has chosen one or the other, augmented with ad-hoc mechanisms to counter their difficulties.

- We define a language that allows programmers to statically restrict the domain of type-directed operations defined in a structural system in a natural manner. Previous work [13, 4] requires that programmers use type-level analysis or programming to makes such restrictions.

- We show how to reconcile **typecase** with the analysis of higher-order type constructors. Previous work [35] has based such analysis on the interpretation of type constructors. In $\lambda_{\mathcal{L}}$, we show how to implement the same operations with simpler constructs.

- We present a sophisticated system of coercions for converting between new types and their definitions. We extend previous work [25, 32, 29] to higher-order coercions.

2. PROGRAMMING IN $\lambda_{\mathcal{L}}$

The $\lambda_{\mathcal{L}}$ language is a polymorphic λ-calculus based on $\mathbb{F}_{\omega}$ [9, 28], augmented with type analysis and user-defined types. The syntax of $\lambda_{\mathcal{L}}$ appears in Figure 1. In addition to the standard kinds, types and terms of $\mathbb{F}_{\omega}$, $\lambda_{\mathcal{L}}$ includes labels, denoted as l, and sets of labels, denoted as $\mathcal{L}$.

Labels may be considered to be "type constants" and model both built-in types, such as int, and names for user-defined types. Labels can be either variables or constants. Arbitrary label constants are taken from an enumerable set and written as ℓ_i^{κ}, annotated by their kind κ and their index i. The requirement that every label constant is associated with a unique index is not necessary for the core language, but is convenient when we add an operator that returns that unique integer associated with the label in Section 4. Some distinguished constants in this language are constructors for primitive types. The label $\ell_0^{\star}$ is a nullary constructor for the type of integers, and $\ell_1^{\star\rightarrow\star\rightarrow\star}$ is the the binary constructor for function types. We use the syntactic sugar ℓ_{int} and $\ell_{\rightarrow}$ to refer to these two labels. However, when these labels appear in types, we use the notation int to stand for ℓ_{int} and $\tau_1 \rightarrow \tau_2$ to stand for the function type $\ell_{\rightarrow}\ \tau_1\ \tau_2$. In the examples, we extend this language with new forms of types, such as booleans (bool), products ($\tau_1 \times \tau_2$), and lists ($\mathsf{list}\ \tau$), and add new label constants, written ℓ_{bool}, $\ell_{\times}$ and ℓ_{list}, to form these types.

Label sets in $\lambda_{\mathcal{L}}$ can be either variables, empty, $\varnothing$, may contain a single label, $\{l\}$, may be the union of two label sets, $\mathcal{L}_1 \cup \mathcal{L}_2$, or may be the entire universe of labels, $\mathcal{U}$. We write $\{l_1, \ldots, l_n\}$ to abbreviate $\{l_1\} \cup \ldots \cup \{l_n\}$.

In the following subsections, we use examples to describe the important features of $\lambda_{\mathcal{L}}$ in more detail.

2.1 Generative types

The $\lambda_{\mathcal{L}}$ language includes a simple mechanism to define new type constants. The expression $\mathbf{new}\ \iota{:}\kappa = \tau\ \mathbf{in}\ e$ creates user-defined labels. This expression dynamically generates a new label constant and binds it to the label variable ι. Inside the scope e, the type ι is isomorphic to the type τ of kind κ. The operators $\{\!\{\cdot\}\!\}^{+}_{\iota=\tau}$ and $\{\!\{\cdot\}\!\}^{-}_{\iota=\tau}$ are available in this scope to coerce expressions to and from the types ι and τ. When τ is apparent from context we elide that annotation, as in the example below.

$$\mathbf{new}\ \iota{:}\star = \mathsf{int}\ \mathbf{in}\ (\lambda x{:}\iota.\{\!\{x\}\!\}^{-}_{\iota} + 3)\{\!\{2\}\!\}^{+}_{\iota}$$

This mechanism dynamically creates new types, so generating these new labels requires an operational effect at run time. However, we have proven [33] that the coercions that convert between a new label and its definition have no run-time cost.

In general, run-time type analysis destroys the parametricity induced by type abstractions. However, $\lambda_{\mathcal{L}}$ can enforce parametric behavior, through a combination of new types and careful scoping. For example, we can be sure that the polymorphic function f below treats its term argument parametrically because, even in the presence of run-time type analysis, it cannot coerce ι to the type int; the definition of f is not in the scope of the new label ι and therefore it cannot contain any analysis of ι nor coercions to and from int.

$$\begin{array}{l}\mathbf{let}\ f = \ldots\ \mathbf{in} \\ \quad \mathbf{new}\ \iota{:}\star = \mathsf{int}\ \mathbf{in}\ f\ [\iota]\ \{\!\{2\}\!\}^{+}_{\iota}\end{array}$$

2.2 Type analysis with a restricted domain

The term $\mathbf{typecase}\ \tau\ e$ can be used to define type-directed operations in $\lambda_{\mathcal{L}}$. This operator determines the head label of the normal form of its type argument τ, such as ℓ_{int}, $\ell_{\times}$, or ℓ_{list}. It then selects the appropriate branch from the finite map e from labels to expressions. For example, the expression $\mathbf{typecase}\ \mathsf{int}\ \{\ell_{\mathsf{int}} \Rightarrow 1, \ell_{\mathsf{bool}} \Rightarrow 2\}$ evaluates to 1.

The finite map in **typecase** may be computed at run time. It is formed from either the empty map $\varnothing$, a singleton map, such as $\{\ell_{\mathsf{int}} \Rightarrow e_{\mathsf{int}}\}$, or the join of two finite maps $e_1 \bowtie e_2$. In a join, if the domains are not disjoint, the rightmost map has precedence and "shadows" any maps to the left with the same domain. Compound maps such as $\{\ell_1 \Rightarrow e_1\} \bowtie \{\ell_2 \Rightarrow e_2\} \bowtie \ldots \bowtie \{\ell_n \Rightarrow e_n\}$ are abbreviated as $\{\ell_1 \Rightarrow e_1, \ell_2 \Rightarrow e_2, \ldots, \ell_n \Rightarrow e_n\}$.

A challenging part of the design of $\lambda_{\mathcal{L}}$ is ensuring that there is a match for the analyzed type. For example, stuck expressions such as **typecase** bool $\{\ell_{\mathsf{int}} \Rightarrow 2\}$ should not type check, because there is no branch for the boolean case.

For this reason, when type checking a **typecase** expression, $\lambda_{\mathcal{L}}$ calculates the set of labels that may appear within the analyzed type and requires that set to be a subset of the set of labels that appear in the domain of the map.

To allow type polymorphism, we annotate a quantified type variable with the set of labels that may appear in types that instantiate it. For example, below we know that α may only be instantiated with a type formed from the labels ℓ_{int} and ℓ_{bool}(i.e., by int or bool), so α will have a match in the **typecase** expression.

$$\Lambda\alpha{:}\star\restriction\{\ell_{\mathsf{int}}, \ell_{\mathsf{bool}}\}.\ \textbf{typecase}\ \alpha\ \{\ell_{\mathsf{int}} \Rightarrow 2, \ell_{\mathsf{bool}} \Rightarrow 3\}$$

If we annotate a type variable with $\mathcal{U}$, the universe of labels, then it is unanalyzable because no **typecase** can cover all branches.

A typical use of **typecase** is polymorphic equality. The function **eq** below implements a polymorphic equality function for data objects composed of integers, booleans, products and lists. In the following, let $\mathcal{L}_0 = \{\ell_{\mathsf{int}}, \ell_{\mathsf{bool}}, \ell_{\times}, \ell_{\mathsf{list}}\}$.

```
$\textbf{fix eq}{:}\forall\alpha{:}\star\restriction\mathcal{L}_0.\ \alpha \to \alpha \to \mathsf{bool}.$
  $\Lambda\alpha{:}\star\restriction\mathcal{L}_0.\ \textbf{typecase}\ \alpha\ \{$
      $\ell_{\mathsf{int}}$    $\Rightarrow$  $\textbf{eqint},$
      $\ell_{\mathsf{bool}}$   $\Rightarrow$  $\lambda x{:}\mathsf{bool}.\ \lambda y{:}\mathsf{bool}.$
                    $\textbf{if}\ x\ \textbf{then}\ y\ \textbf{else}\ (\textbf{not}\ y),$
      $\ell_{\times}$      $\Rightarrow$  $\Lambda\alpha_1{:}\star\restriction\mathcal{L}_0.\ \Lambda\alpha_2{:}\star\restriction\mathcal{L}_0.$
                    $\lambda x{:}(\alpha_1 \times \alpha_2).\ \lambda y{:}(\alpha_1 \times \alpha_2).$
                      $\textbf{eq}[\alpha_1](\textbf{fst}\,x)(\textbf{fst}\,y)$
                        $\&\&\ \textbf{eq}[\alpha_2](\textbf{snd}\,x)(\textbf{snd}\,y),$
      $\ell_{\mathsf{list}}$   $\Rightarrow$  $\Lambda\beta{:}\star\restriction\mathcal{L}_0.\ \lambda x{:}(\mathsf{list}\ \beta).\ \lambda y{:}(\mathsf{list}\ \beta).$
                    $\textbf{all2}\ (\textbf{eq}[\beta])\ x\ y$
  $\}$
```

Product types have two subcomponents, so the branch for $\ell_{\times}$ abstracts two type variables for those subcomponents. Likewise, the ℓ_{list} case abstracts the type of list elements. In general, the type of each branch in a **typecase** is determined by the kind of the matched label. After **typecase** determines the head label of its argument, it steps to the corresponding branch and applies that branch to any type arguments that were applied to the head label. For example, applying polymorphic equality to the type of integer lists results in the ℓ_{list} branch being applied to int.

```
$\textbf{eq}[\mathsf{list\ int}] \mapsto (\Lambda\beta{:}\star\restriction\mathcal{L}_0.\ \lambda x{:}(\mathsf{list}\ \beta).\ \lambda y{:}(\mathsf{list}\ \beta).$
                  $\textbf{all2}\ (\textbf{eq}[\beta])\ x\ y)\ \ [\mathsf{int}]$
            $\mapsto \lambda x{:}(\mathsf{list\ int}).\ \lambda y{:}(\mathsf{list\ int}).\ \textbf{all2}\ (\textbf{eq}[\mathsf{int}])\ x\ y$
```

The ability to restrict the arguments of a polytypic function is valuable. For example, the polytypic equality function cannot be applied to values of function type. Here, $\lambda_{\mathcal{L}}$ naturally makes this restriction by omitting $\ell_{\to}$ from $\mathcal{L}_0$.

2.3 Generative types and type analysis

The function **eq** is closed to extension. However, with the creation of new labels there may be a large universe of types of expressions that programmers would like to apply **eq** to.

In $\lambda_{\mathcal{L}}$, we provide two solutions to this problem. We can rewrite **eq** to be extensible with new branches for the new labels. Otherwise, we can leave **eq** as it is and at each call site, coerce all the arguments to **eq** so that their types do not contain new labels. Furthermore, programmers may choose a combination of these two solutions.

2.3.1 Extensible type analysis

In $\lambda_{\mathcal{L}}$, we can write a version of **eq** that can be extended with new branches for new labels. Programmers may provide new **typecase** branches as an additional argument to **eq**. The type of this argument, a first-class map from labels to expressions, is written as $\mathcal{L}_1 \Rrightarrow \tau \restriction \mathcal{L}_2$. The first component, $\mathcal{L}_1$, is the domain of the map. The types of the expressions in the range of the map are determined by τ, $\mathcal{L}_2$, and the kinds of labels in $\mathcal{L}_1$. Some of these types can be polymorphic and the label set $\mathcal{L}_2$ records the restrictions on the quantified variables of these types. We present a more detailed explanation of the semantics of maps in Section 3.

Using first-class maps, we can supply a branch for int's into the following operation:

$$\lambda x{:}(\{\ell_{\mathsf{int}}\} \Rrightarrow (\lambda\alpha{:}\star.\mathsf{bool}) \restriction \{\ell_{\mathsf{int}}, \ell_{\mathsf{bool}}\}).$$
$$\textbf{typecase}\ \mathsf{int}\ (\{\ell_{\mathsf{bool}} \Rightarrow \textbf{true}\} \bowtie x)$$

In the above example, the argument x represents a branch that can handle the label ℓ_{int}. We postpone the detailed discussion of the type annotation of x until Section 3.

Because existing maps may be shadowed in joins, type directed operations can be possibly redefined for those names that belong in the domain of the maps that get shadowed.

However, even if a type-directed function abstracts a map for **typecase**, it is still not extensible. The type of that map specifies the labels that are in its domain. Branches for newly created labels cannot be supplied.

Therefore, $\lambda_{\mathcal{L}}$ includes *label set polymorphism.* A typical idiom for an extensible operation is to abstract a set of labels, a map for that set, and then require that the argument to the polytypic function be composed of those labels plus any labels that already have branches in **typecase**. For example, let us create an open version of **eq**. Let $\mathcal{L}_0 = \{\ell_{\mathsf{int}}, \ell_{\mathsf{bool}}\}$. In the code below, s describes the domain of labels in the map y. The **eq** function may be instantiated with types containing labels from $\mathcal{L}_0$ or s.

```
$\textbf{eq} = \Lambda s{:}\mathrm{Ls}.\lambda x{:}(s \Rrightarrow (\lambda\alpha{:}\star.\alpha \to \alpha \to \mathsf{bool}) \restriction s \cup \mathcal{L}_0).$
  $\textbf{fix eq}{:}\forall\alpha{:}\star\restriction(s \cup \mathcal{L}_0).\ \alpha \to \alpha \to \mathsf{bool}.$
    $\Lambda\alpha{:}\star\restriction(s \cup \mathcal{L}_0).\ \textbf{typecase}\ \alpha\ x \bowtie \{$
      $\ell_{\mathsf{int}}$    $\Rightarrow$  $\textbf{eqint},$
      $\ell_{\mathsf{bool}}$   $\Rightarrow$  $\lambda y{:}\mathsf{bool}.\ \lambda z{:}\mathsf{bool}.$
                    $\textbf{if}\ y\ \textbf{then}\ z\ \textbf{else}\ (\textbf{not}\ z)\}$
```

Here we use $\Lambda s{:}\mathrm{Ls}$ for label set abstraction. Now, we write an extension to **eq** for products. The extension will need to call the extended version of **eq** for the components of the products, hence it will have to be recursive.

```
$\textbf{ext} = \textbf{fix ext}{:}\ell_{\times} \Rrightarrow (\lambda\alpha{:}\star.\alpha \to \alpha \to \mathsf{bool}) \restriction \mathcal{L}_0 \cup \{\ell_{\times}\}.$
    $\{\ \ell_{\times} \Rightarrow \Lambda\alpha_1{:}\star\restriction\mathcal{L}_0 \cup \{\ell_{\times}\}.\ \Lambda\alpha_2{:}\star\restriction\mathcal{L}_0 \cup \{\ell_{\times}\}.$
              $\lambda x{:}(\alpha_1 \times \alpha_2).\ \lambda y{:}(\alpha_1 \times \alpha_2).$
                $\textbf{eq}[\{\ell_{\times}\}]\textbf{ext}[\alpha_1](\textbf{fst}\,x)(\textbf{fst}\,y)$
                  $\&\&\ \textbf{eq}[\{\ell_{\times}\}]\textbf{ext}[\alpha_2](\textbf{snd}\,x)(\textbf{snd}\,y)\}$
```

We call the extended equality function as follows.

$$\mathbf{eq}\ [\{\ell_\times\}]\ \mathbf{ext}\ [\mathsf{int} \times \mathsf{bool}]\ (1, \mathbf{false})\ (2, \mathbf{true})$$

This calculus explicitly witnesses the design complexity of open polytypic operations. Suppose we wished to call an open operation, called **important**, in the body of an open serializer, called **tostring**. Intuitively, **important** elides part of a data structure by deciding whether recursion should continue. Because **tostring** can be applied to any type that provides a map for new labels, **important** must also be applicable to all those types. There are two ways to write **tostring**. The first is to supply the branches for **important** as an additional argument to **tostring**, as below.

$$\begin{array}{l}
\Lambda s{:}\mathrm{LS}.\lambda y_{tos}{:}(s{\Rightarrow}(\lambda\alpha{:}\star.\alpha \to \mathsf{string}){\upharpoonright}s \cup \{\ell_\times\}).\\
\quad \lambda y_{imp}{:}(s{\Rightarrow}(\lambda\alpha{:}\star.\alpha \to \mathsf{string}){\upharpoonright}s \cup \{\ell_\times\}).\\
\quad\quad \mathbf{fix\ tostring}.\ \Lambda\alpha{:}(\star{\upharpoonright}s \cup \{\ell_\times\}).\\
\quad\quad\quad \mathbf{typecase}\ \alpha\\
\quad\quad\quad\quad (y_{tos} \bowtie \{\ell_\times \Rightarrow\\
\quad\quad\quad\quad\quad \Lambda\alpha_1{:}\star{\upharpoonright}(s \cup \{\ell_\times\}).\ \Lambda\alpha_2{:}\star{\upharpoonright}(s \cup \{\ell_\times\}).\\
\quad\quad\quad\quad\quad\quad \lambda x{:}(\alpha_1 \times \alpha_2).\\
\quad\quad\quad\quad\quad\quad\quad \mathbf{let}\ s1 = \ \mathbf{if\ important}[s]\ y_{imp}\ [\alpha_1](\mathbf{fst}\,x)\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{then\ tostring}[s][\alpha_1](\mathbf{fst}\,x)\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{else}\ \text{“...”}\ \mathbf{in}\\
\quad\quad\quad\quad\quad\quad\quad \mathbf{let}\ s2 = \mathbf{if\ important}[s]\ y_{imp}\ [\alpha_2](\mathbf{snd}\,x)\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{then\ tostring}[s][\alpha_2](\mathbf{snd}\,x)\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{else}\ \text{“...”}\ \mathbf{in}\\
\quad\quad\quad\quad\quad\quad \text{“(”}\ +\!\!+\ s1\ +\!\!+\ \text{“,”}\ s2\ +\!\!+\ \text{“)”}\})
\end{array}$$

Dependency-Style Generic Haskell [22] uses this technique. In that language, the additional arguments are automatically inferred by the compiler. However, the dependencies still show up in the type of an operation, hindering the modularity of the program.

A second solution is to provide to **tostring** a mechanism for coercing away the labels in the set s before the call to **important**. In that case, **important** would not be able to specialize its execution to the newly provided labels. However, if **tostring** called many open operations, or if it were somehow infeasible to supply a map for **important**, then that may be the only reasonable implementation. In contrast, a *closed* polytypic operation may easily call other *closed* polytypic functions.

2.3.2 Higher-order coercions

In some cases, such as structural equality, the behavior of a type-directed operation for a new user-defined type should be identical to that for its underlying definition. However, it is in general computationally expensive to coerce the components of a large data structure, using the coercion mechanism described earlier.

Consider the following example. Suppose that we define a new label isomorphic to a pair of integers with $\mathbf{new}\ \iota{:}\star = \mathsf{int} \times \mathsf{int}$ and let x be a variable of type $\mathsf{list}\ \iota$. Say also that we have a closed, type-directed operation f of type $\forall\alpha{:}\star{\upharpoonright}\{\ell_{\mathsf{int}}, \ell_\times, \ell_\to, \ell_{\mathsf{list}}\}.\ \alpha \to \mathsf{int}$. The call $f\ [\mathsf{list}\ \iota]\ x$ does not type check because ι is not in the domain of f. Since we know that ι is isomorphic to $\mathsf{int} \times \mathsf{int}$, we could call f after coercing the type of the elements of the list by mapping the first-order coercion across the list.

$$f\ [\mathsf{list}\ \mathsf{int} \times \mathsf{int}]\ (\mathbf{map}\ (\lambda y{:}\iota.\ \{\!|y|\!\}_\iota^-)\ x)$$

However, operationally this map destructs and rebuilds the list, which is computationally expensive. Instead, $\lambda_{\mathcal{L}}$ also includes *higher-order coercions*, which can coerce x to be of type $\mathsf{list}\ \mathsf{int} \times \mathsf{int}$ without computational cost.

$$f\ [\mathsf{list}\ \mathsf{int} \times \mathsf{int}]\ \{\!|x : \mathsf{list}|\!\}_\iota^-$$

Higher-order coercions have no run-time effect; they merely alter the types of expressions. For reasons of type checking, a higher-order coercion is annotated with a type constructor—in this case list—that pinpoints the location of the label to coerce in the type of the term.

2.4 Higher-order type analysis

Higher-order type analysis [35] is often used to define operations in terms of parameterized data structures, such as *lists* and *trees*. These operations must be able to distinguish between the type parameter and the rest of the type.

We present a characteristic example, due to Hinze [14]. Let $\mathcal{L}_0 = \{\ell_{\mathsf{int}}, \ell_{\mathsf{bool}}, \ell_\times\}$ and consider the following open function:

$$\begin{array}{l}
\mathbf{ecount} = \Lambda s{:}\mathrm{LS}.\lambda ext{:}(s{\Rightarrow}(\lambda\alpha{:}\star.\alpha \to \alpha \to \mathsf{int}){\upharpoonright}s \cup \mathcal{L}_0).\\
\quad \mathbf{fix\ ecount}{:}\forall\alpha{:}\star{\upharpoonright}s \cup \mathcal{L}_0.\ \alpha \to \alpha \to \mathsf{int}.\\
\quad\quad \Lambda\alpha{:}\star{\upharpoonright}s \cup \mathcal{L}_0.\ \mathbf{typecase}\ \alpha\ ext \bowtie \{\\
\quad\quad\quad\quad \begin{array}{lcl}
\ell_{\mathsf{int}} & \Rightarrow & \lambda x{:}\mathsf{int}.\ 0,\\
\ell_{\mathsf{bool}} & \Rightarrow & \lambda x{:}\mathsf{bool}.\ 0,\\
\ell_\times & \Rightarrow & \Lambda\alpha_1{:}\star{\upharpoonright}s \cup \mathcal{L}_0.\ \Lambda\alpha_2{:}\star{\upharpoonright}s \cup \mathcal{L}_0.\\
 & & \quad \lambda x{:}\alpha_1 \times \alpha_2.\\
 & & \quad\quad (\mathbf{ecount}[\alpha_1](\mathbf{fst}\ x)) +\\
 & & \quad\quad (\mathbf{ecount}[\alpha_2](\mathbf{snd}\ x))\ \}
\end{array}
\end{array}$$

The function might seem useless, as it returns 0 for every type created by integers, booleans and products. However, suppose that we have a type constructor τ, of kind $\star \to \star$:

$$\tau = \lambda\alpha{:}\star.(\alpha \times \mathsf{int}) \times \alpha$$

We would like to be able to count the number of data values of type α in an instance of the structure $\tau\ \alpha$. In this example, the answer is always 2, but in more complex data structures, such as lists or trees, the answer will be the actual length of the list or the number of nodes in the tree. We need to be able to distinguish between the data—no matter what type it is—and the rest of the structure. Because $\lambda_{\mathcal{L}}$ can generate new labels at run time, it can make such distinctions. Consider the following function:

$$\begin{array}{l}
\mathbf{size} = \Lambda\gamma{:}\star \to \star{\upharpoonright}\mathcal{L}_0.\ \Lambda\alpha{:}\star{\upharpoonright}\mathcal{L}_0.\\
\quad \mathbf{new}\ \iota{:}\star = \alpha\ \mathbf{in}\\
\quad\quad \lambda x : \gamma\ \alpha.\ \mathbf{ecount}[\{\iota\}]\\
\quad\quad\quad\quad \{\iota \ \Rightarrow\ \lambda x{:}\iota.\ 1\}\ [\gamma\ \iota]\ \{\!|x : \gamma|\!\}_\iota^-
\end{array}$$

The function abstracts the constructor γ and its type parameter α. It then creates a new label ι, isomorphic to α, and takes an argument of type $\gamma\ \alpha$. The argument type is coerced to type $\gamma\ \iota$ and, finally, the function **ecount** is applied to the argument, taking an extension for label ι that returns 1. The function will effectively compute the number of data nodes in our data structure. The following will return 2.

$$\mathbf{size}\ [\tau][\mathsf{int}]((1, 2), 2)$$

The reader is invited to study the examples included in the implementation.

Static Judgments	
Kind well-formedness	$\Delta \vdash \kappa$
Label well-formedness	$\Delta \vdash l : \mathrm{L}(\kappa)$
Label set well-formedness	$\Delta \vdash \mathcal{L} : \mathrm{Ls}$
Label set subsumption	$\Delta \vdash \mathcal{L}_1 \sqsubseteq \mathcal{L}_2$
Label set equivalence	$\Delta \vdash \mathcal{L}_1 = \mathcal{L}_2$
Type well-formedness	$\Delta \vdash \tau : \kappa$
Type label set analysis	$\Delta \vdash \tau \,\vert\, \mathcal{L}$
Type equivalence	$\Delta \vdash \tau_1 = \tau_2 : \kappa$
Term well-formedness	$\Delta; \Gamma \vdash e : \tau \,\vert\, \Sigma$
Dynamic Judgments	
Small-step evaluation	$\mathcal{L}; e \mapsto \mathcal{L}'; e'$
Weak-head reduction	$\Delta \vdash \tau \Downarrow \tau'$
Weak-head normalization	$\Delta \vdash \tau \downarrow \tau'$
Path conversion	$\rho \rightsquigarrow p$

Figure 2: Core $\lambda_{\mathcal{L}}$ judgments

$$\begin{array}{lrcl}
\textit{Type Contexts} & \Delta & ::= & \cdot \mid \Delta, \iota{:}\mathrm{L}(\kappa) \mid \Delta, s{:}\mathrm{Ls} \\
 & & \mid & \Delta, \alpha{:}\kappa{\restriction}\mathcal{L} \\
\textit{Type isomorphisms} & \Sigma & ::= & \cdot \mid \Sigma, l{:}\kappa = \tau \\
\textit{Term Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \\
\textit{Values} & v & ::= & \lambda x{:}\sigma.e \mid i \mid \{\!\!\{v\}\!\!\}^{+}_{l=\tau} \\
 & & \mid & \varnothing \mid \{l \Rightarrow e\} \mid v_1 \bowtie v_2 \\
 & & \mid & \Lambda\alpha{:}\kappa{\restriction}\mathcal{L}.e \mid \Lambda\iota{:}\mathrm{L}(\kappa).e \mid \Lambda s{:}\mathrm{Ls}.e \\
\textit{Type paths} & \rho & ::= & \bullet \mid \rho\ \tau \\
\textit{Term paths} & p & ::= & \bullet \mid p\ [\tau]
\end{array}$$

Figure 3: Additional syntactic categories

3. THE CORE LANGUAGE

Next we describe the semantics of core $\lambda_{\mathcal{L}}$ in detail. Figure 3 contains additional syntactic categories necessary for the presentation of the dynamic and static semantics. Type and term contexts are standard, but note that type contexts also record the label set restrictions of type variables. Type isomorphisms Σ are used to record the isomorphisms between labels and types that are introduced by **new** expressions. Type paths ρ are simply type-level applications of a hole, $\bullet$, to a sequence of types, and term paths p are term-level applications of a hole to a sequence of types. The judgments of the language are given in Figure 2. In this section and the next section we present the most important judgments. The full semantics of the language can be found in the technical report [33].

The judgment $\Delta; \Gamma \vdash e : \sigma \,|\, \Sigma$ states that a term e is well-formed with type σ, in type context Δ, term context Γ, and possibly using type isomorphisms in Σ. To show that terms are well typed often requires determining the kinds of types, with the judgment $\Delta \vdash \tau : \kappa$, and the set of possible labels that may appear in types, with the judgment $\Delta \vdash \tau \,|\, \mathcal{L}$.

The judgment $\mathcal{L}; e \mapsto \mathcal{L}'; e'$ describes the small-step call-by-value operational semantics of the language. A term e with a set of labels $\mathcal{L}$ steps to a new term e' possibly with larger set of labels $\mathcal{L}'$. During the evaluation of the **new** operator, a fresh label constant is generated and added to the label set component. In this way, it resembles an *allocation semantics* [24, 10]. The initial state of execution includes the label constants ℓ_{int} and $\ell_{\rightarrow}$ in $\mathcal{L}$. The semantics for the λ-calculus fragment of $\lambda_{\mathcal{L}}$, including **fix** and integers, is standard, so we will not discuss it further.

3.1 Semantics of generative types

The dynamic and static rules for **new** are:

$$\frac{\ell_i^{\kappa} \notin \mathcal{L}}{\mathcal{L}; \textbf{new}\ \iota{:}\kappa = \tau\ \textbf{in}\ e \mapsto \mathcal{L} \cup \{\ell_i^{\kappa}\}; e[\ell_i^{\kappa}/\iota]}$$

$$\frac{\Delta, \iota{:}\mathrm{L}(\kappa); \Gamma \vdash e : \sigma \,|\, \Sigma, \iota{:}\kappa = \tau \qquad \Delta, \iota{:}\mathrm{L}(\kappa) \vdash \tau : \kappa \qquad \iota \notin \sigma}{\Delta; \Gamma \vdash \textbf{new}\ \iota{:}\kappa = \tau\ \textbf{in}\ e : \sigma \,|\, \Sigma}$$

Dynamically, the **new** operation chooses a label constant that has not been previously referred to and substitutes it for the label variable ι within the scope of e. Statically, ι must not appear in the type σ of e, so that it does not escape its scope. Recursive definitions of new labels are allowed. When type checking e, the isomorphism between ι and τ is available through the coercions.

The primitive coercions change the head constructor in the type of their arguments.

$$\frac{\Delta; \Gamma \vdash e : \rho[\tau] \,|\, \Sigma \qquad l{:}\kappa = \tau \in \Sigma}{\Delta; \Gamma \vdash \{\!\!\{e\}\!\!\}^{+}_{l=\tau} : \rho[l] \,|\, \Sigma}$$

$$\frac{\Delta; \Gamma \vdash e : \rho[l] \,|\, \Sigma \qquad l{:}\kappa = \tau \in \Sigma}{\Delta; \Gamma \vdash \{\!\!\{e\}\!\!\}^{-}_{l=\tau} : \rho[\tau] \,|\, \Sigma}$$

The syntax $\rho[\tau]$ denotes a type where τ is the head of the type path ρ. Operationally, the primitive coercion $\{\!\!\{\cdot\}\!\!\}^{-}_{l=\tau}$ cancels the primitive coercion $\{\!\!\{\cdot\}\!\!\}^{+}_{l=\tau}$.

$$\frac{}{\mathcal{L}; \{\!\!\{\{\!\!\{v\}\!\!\}^{+}_{l=\tau}\}\!\!\}^{-}_{l=\tau} \mapsto \mathcal{L}; v}$$

Higher-order coercions allow the non-head positions of a type to change. These coercions are annotated with a type constructor τ' that describes the part of the data structure to be coerced. A higher-order coercion "maps" the primitive coercions over an expression, guided by the weak-head normal form of the type constructor τ'.

$$\frac{\Delta; \Gamma \vdash e : \tau'\ \tau \,|\, \Sigma \qquad l{:}\kappa = \tau \in \Sigma}{\Delta; \Gamma \vdash \{\!\!\{e : \tau'\}\!\!\}^{+}_{l=\tau} : \tau'\ l \,|\, \Sigma}$$

$$\frac{\Delta; \Gamma \vdash e : \tau'\ l \,|\, \Sigma \qquad l{:}\kappa = \tau \in \Sigma}{\Delta; \Gamma \vdash \{\!\!\{e : \tau'\}\!\!\}^{-}_{l=\tau} : \tau'\ \tau \,|\, \Sigma}$$

The weak-head normal form is determined through the following kind-directed judgment:

$$\frac{\Delta \vdash \tau : \star \qquad \Delta \vdash \tau \Downarrow_{*} \tau' \qquad \Delta \vdash \tau' \not\Downarrow}{\Delta \vdash \tau \downarrow \tau'}$$

$$\frac{\Delta \vdash \tau : \kappa_1 \rightarrow \kappa_2 \qquad \Delta, \alpha{:}\kappa_1 \vdash \tau\ \alpha \downarrow \tau'}{\Delta \vdash \tau \downarrow \lambda\alpha{:}\kappa_1.\tau}$$

The first rule assures that if a type is of kind $\star$, then it normalizes to its weak-head normal form. The judgment $\Delta \vdash \tau \Downarrow \tau'$ is a standard weak-head reduction relation. If a type is not of kind $\star$ the second rule applies, so that eventually it will reduce to a nesting of abstractions around a weak-head normal form.

$$\frac{\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\rho[\alpha]}{\mathcal{L}; \{\!\{v : \tau'\}\!\}^{+}_{l=\tau} \mapsto \mathcal{L}; \{\!\{\{\!\{v : \lambda\alpha{:}\kappa.\rho[\tau]\}\!\}^{+}_{l=\tau}\}\!\}^{+}_{l=\tau}} \qquad \frac{\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\rho[\alpha]}{\mathcal{L}; \{\!\{v : \tau'\}\!\}^{-}_{l=\tau} \mapsto \mathcal{L}; \{\!\{\{\!\{v : \lambda\alpha{:}\kappa.\rho[l]\}\!\}^{-}_{l=\tau}\}\!\}^{-}_{l=\tau}} \qquad \frac{\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\ell_{\mathsf{int}}}{\mathcal{L}; \{\!\{i : \tau'\}\!\}^{\pm}_{l=\tau} \mapsto \mathcal{L}; i}$$

$$\frac{\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\tau_1 \to \tau_2 \qquad \vdash \tau_1' = \tau_1[\tau/\alpha] : \star}{\mathcal{L}; \{\!\{\lambda x{:}\tau_1'.e : \tau'\}\!\}^{+}_{l=\tau} \mapsto \mathcal{L}; \lambda x{:}(\tau_1[l/\alpha]).\{\!\{e[\{\!\{x : \lambda\alpha{:}\kappa.\tau_1\}\!\}^{-}_{l=\tau}/x] : \lambda\alpha{:}\kappa.\tau_2\}\!\}^{+}_{l=\tau}} \qquad \frac{\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\mathcal{L}_1 \Rightarrow \tau' \upharpoonright \mathcal{L}_2}{\mathcal{L}; \{\!\{\varnothing : \tau'\}\!\}^{\pm}_{l=\tau} \mapsto \mathcal{L}; \varnothing}$$

$$\frac{\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\mathcal{L}_1 \cup \mathcal{L}_2 \Rightarrow \tau' \upharpoonright \mathcal{L}}{\mathcal{L}; \{\!\{v_1 \bowtie v_2 : \tau'\}\!\}^{\pm}_{l=\tau} \mapsto \mathcal{L}; \{\!\{v_1 : \lambda\alpha{:}\kappa.\mathcal{L}_1 \Rightarrow \tau' \upharpoonright \mathcal{L}\}\!\}^{\pm}_{l=\tau} \bowtie \{\!\{v_2 : \lambda\alpha{:}\kappa.\mathcal{L}_2 \Rightarrow \tau' \upharpoonright \mathcal{L}\}\!\}^{\pm}_{l=\tau}}$$

$$\frac{\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\rho[l_1]}{\mathcal{L}; \{\!\{\{\!\{v\}\!\}^{+}_{l_1=\tau_1} : \tau'\}\!\}^{\pm}_{l_2=\tau_2} \mapsto \mathcal{L}; \{\!\{\{\!\{v : \lambda\alpha{:}\kappa.\rho[\tau_1]\}\!\}^{\pm}_{l_2=\tau_2}\}\!\}^{+}_{l_1=\tau_1}}$$

Figure 4: Operational semantics for higher-order coercions (excerpt)

Figure 4 lists some of the rules for higher-order coercions. Higher order coercions first reduce the type constructor τ', using the relation described above. Because the type constructor annotation τ' on a higher-order coercion must be of kind $\kappa \to \star$ for some kind κ, we know that it will reduce to a type constructor of the form $\lambda\alpha{:}\kappa.\tau$. We also know that τ will be a path headed by a variable or constant, a universal type, or a branch type. The form of τ determines the execution of the coercion.

If τ is a path beginning with a type variable α, then that is a location where a first-order coercion should be used. However, there may be other parts of the value that should be coerced—there may be other occurrences of α in the path besides the head position—so inside the first-order coercion is another higher-order coercion.

Otherwise the form of τ must match the value in the body of the coercion. For each form of type there is an operational rule. For example, if τ is int then the value must be an integer, and the coercion goes away—no primitive coercions are necessary. If τ is a function type, then the semantics pushes the coercion through the function, changing the type of its argument and the body of the function. Similar rules apply to other type forms.

3.2 Semantics of type analysis

We next describe the semantics of type analysis. The rule for the execution of **typecase** is below:

$$\frac{\vdash \tau \downarrow \rho[\ell_i^\kappa] \qquad \{\ell_i^\kappa \Rightarrow e'\} \in v \qquad \rho \rightsquigarrow p}{\mathcal{L}; \textbf{typecase}\ \tau\ v \mapsto \mathcal{L}; p[e']}$$

This rule uses the judgment $\Delta \vdash \tau \downarrow \tau'$ to determine the weak-head normal form of the analyzed type τ. This form must be some label ℓ_i^κ at the head of a type path ρ. Then, **typecase** chooses the rightmost matching branch from its map argument, v, and steps to the specified term, applying the same series of type arguments as in the type path ρ. (The term path p is derived from the type path ρ in the obvious fashion.

The static semantics of **typecase** is defined by the following rule.

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash e : \mathcal{L}_1 \Rightarrow \tau' \upharpoonright \mathcal{L}_2 \Sigma \\ \Delta \vdash \tau : \star \qquad \Delta \vdash \tau \,|\, \mathcal{L} \qquad \Delta \vdash \mathcal{L}_1 \sqsubseteq \mathcal{L}_2 \qquad \Delta \vdash \mathcal{L} \sqsubseteq \mathcal{L}_1\end{array}}{\Delta; \Gamma \vdash \textbf{typecase}\ \tau\ e : \tau'\tau \,|\, \Sigma}$$

The most important part of this rule is that it checks that τ may be safely analyzed by **typecase**. There must be a corresponding branch in **typecase** for any label that may be the head of the normal form of τ. The judgment $\Delta \vdash \tau \,|\, \mathcal{L}$ conservatively determines the set of labels that could appear as part of the type τ. This judgment states that in the typing context Δ, the type τ may mention labels in the set $\mathcal{L}$. The rules for this judgment are given below.

$$\frac{}{\Delta \vdash l \,|\, \{l\}} \qquad \frac{\alpha{:}\kappa \upharpoonright \mathcal{L} \in \Delta}{\Delta \vdash \alpha \,|\, \mathcal{L}} \qquad \frac{\Delta, \alpha{:}\kappa_1 \upharpoonright \varnothing \vdash \tau \,|\, \mathcal{L}}{\Delta \vdash \lambda\alpha{:}\kappa_1.\tau \,|\, \mathcal{L}}$$

$$\frac{\Delta \vdash \tau_1 \,|\, \mathcal{L}_1 \qquad \Delta \vdash \tau_2 \,|\, \mathcal{L}_2}{\Delta \vdash \tau_1\tau_2 \,|\, \mathcal{L}_1 \cup \mathcal{L}_2}$$

In the first rule, labels are added to the set when they are used as types. The second rule corresponds to type variables bound in the context. These variables are annotated with the set of labels that may appear in types that are used to instantiate them. The third rule shows that variables bound by type-level abstractions are annotated with the empty label set. This rule is sound because the appropriate labels will be recorded when the type-level abstraction is applied. Finally, the label set of a type application is the union of the label sets of the the type function and the argument type.

Not all types are analyzable in core $\lambda_{\mathcal{L}}$. The types of first-class maps and polymorphic expressions may not be analyzed because they do not have normal forms that have labels at their heads. We prevent such types from being the argument to **typecase** by not including rules to determine a label set for those types. In the next section, we show how to extend the calculus so that such types may be represented by labels, and therefore analyzed.

Once the rule for type checking **typecase** determines the labels that could appear in the argument type, it looks at the type of the first-class map to determine the domain of the map. Given some map e with domain $\mathcal{L}_1$ and a type argument τ that mentions labels in $\mathcal{L}$, the rule checks that the map can handle all possible labels in τ with $\Delta \vdash \mathcal{L} \sqsubseteq \mathcal{L}_1$. (This judgment extends set inclusion to reason about label and label set variables.)

We discuss the reason for the $\Delta \vdash \mathcal{L}_1 \sqsubseteq \mathcal{L}_2$ premise below. Finally, the result type of **typecase** depends on τ from the type of the map argument.

A number of rules check the formation of first-class maps. The most important rule is the rule for singleton maps below.

$$\frac{\Delta \vdash \mathcal{L} : \mathrm{Ls} \qquad \Delta \vdash l : \mathrm{L}(\kappa) \qquad \Delta; \Gamma \vdash e : \tau'\langle l : \kappa \upharpoonright \mathcal{L}\rangle \mid \Sigma}{\Delta; \Gamma \vdash \{l \Rightarrow e\} : \{l\} \Rightarrow \tau' \upharpoonright \mathcal{L} \mid \Sigma}$$

The first component of the map type (in this case l) describes the domain of the map and the second two components (τ' and $\mathcal{L}'$) describe the types of the branches of the map. The judgments $\Delta \vdash l : \mathrm{L}(\kappa)$ and $\Delta \vdash \mathcal{L} : \mathrm{Ls}$ ensure that the label l and label set $\mathcal{L}$ are well-formed with respect to the type context Δ. For labels of higher kind, **typecase** applies the matching branch to all of the arguments in the path to the matched label. Therefore, the branch for that label must quantify over all of those arguments. The correct type for this branch is determined by the kind of the label, with the polykinded type notation $\tau'\langle \tau : \kappa \upharpoonright \mathcal{L}\rangle$. This notation is defined by the following rules:

$$\begin{array}{lcl} \tau'\langle \tau : \star \upharpoonright \mathcal{L}\rangle & \triangleq & \tau'\ \tau \\ \tau'\langle \tau : \kappa_1 \to \kappa_2 \upharpoonright \mathcal{L}\rangle & \triangleq & \forall \alpha{:}\kappa_1 \upharpoonright \mathcal{L}.\tau'\langle \tau\ \alpha : \kappa_2 \upharpoonright \mathcal{L}\rangle \end{array}$$

The label set component of this kind-indexed type is used as the restriction for the quantified type variables. To ensure that it is safe to apply the branch to the subcomponents of the type argument, the rule for **typecase** requires that the second label set in the type of the map be at least as big as the first label set.

The rule for checking compound maps is the following.

$$\frac{\Delta; \Gamma \vdash e_1 : \mathcal{L}_1 \Rightarrow \tau' \upharpoonright \mathcal{L} \mid \Sigma \qquad \Delta; \Gamma \vdash e_2 : \mathcal{L}_2 \Rightarrow \tau' \upharpoonright \mathcal{L} \mid \Sigma}{\Delta; \Gamma \vdash e_1 \bowtie e_2 : \mathcal{L}_1 \cup \mathcal{L}_2 \Rightarrow \tau' \upharpoonright \mathcal{L} \mid \Sigma}$$

The domain of the compound map is the union of the two domains. However the two maps must have the same type component and the same label set that restricts the quantified type variables in the range of the maps. On the other hand empty branches can be typed using any type component and restriction label set.

$$\frac{\Delta \vdash \mathcal{L} : \mathrm{Ls} \qquad \Delta \vdash \tau' : \star \to \star}{\Delta; \Gamma \vdash \varnothing : \varnothing \Rightarrow \tau' \upharpoonright \mathcal{L} \mid \Sigma}$$

3.2.1 A digression on branch types

Here we give some examples on the semantics of map types. Consider the branch from the vanilla equality function in Section 2 that corresponds to products.

$$\begin{array}{l} \{\ell_\times \Rightarrow \Lambda\alpha_1{:}\star\upharpoonright\mathcal{L}_0.\ \Lambda\alpha_2{:}\star\upharpoonright\mathcal{L}_0. \\ \qquad \lambda x{:}(\alpha_1 \times \alpha_2).\ \lambda y{:}(\alpha_1 \times \alpha_2). \\ \qquad\quad \mathbf{eq}[\alpha_1](\mathbf{fst}\,x)(\mathbf{fst}\,y) \\ \qquad\qquad \&\&\ \mathbf{eq}[\alpha_2](\mathbf{snd}\,x)(\mathbf{snd}\,y)\} \end{array}$$

Let us abbreviate the body of the branch as e. According to the rule for checking branches, there must be a type τ', and a label set $\mathcal{L}$, such that the following holds:

$$\Delta; \Gamma \vdash \{\ell_\times \Rightarrow e\} : \{\ell_\times\} \Rightarrow \tau' \upharpoonright \mathcal{L} \mid \cdot$$

Above, Γ must hold the typing assumption for the recursive function **eq** and Δ must contain the type variable α. Take $\mathcal{L} = \mathcal{L}_0$ and $\tau' = \lambda\alpha{:}\star.\alpha \to \alpha \to \mathsf{bool}$. We need to confirm that:

$$\Delta; \Gamma \vdash e : \tau'\langle \ell_\times : \star \to \star \to \star \upharpoonright \mathcal{L}\rangle \mid \cdot$$

Using the equivalence rules for polykinded types we get (where "=" should be interpreted as a standard type equivalence relation):

$$\begin{array}{ll} & \tau'\langle \ell_\times : \star \to \star \to \star \upharpoonright \mathcal{L}\rangle \\ = & \forall\alpha_1{:}\star\upharpoonright\mathcal{L}_0.\tau'\langle \ell_\times\ \alpha_1 : \star \to \star \upharpoonright \mathcal{L}_0\rangle \\ = & \forall\alpha_1{:}\star\upharpoonright\mathcal{L}_0.\forall\alpha_2{:}\star\upharpoonright\mathcal{L}_0.\tau'\langle \ell_\times\ \alpha_1\ \alpha_2 : \star \upharpoonright \mathcal{L}_0\rangle \\ = & \forall\alpha_1{:}\star\upharpoonright\mathcal{L}_0.\forall\alpha_2{:}\star\upharpoonright\mathcal{L}_0.(\ell_\times\ \alpha_1\ \alpha_2) \to (\ell_\times\ \alpha_1\ \alpha_2) \to \mathsf{bool} \end{array}$$

This is the correct type for e.

The rôle of $\mathcal{L}_2$ in a branch type $\mathcal{L}_1 \Rightarrow \tau' \upharpoonright \mathcal{L}_2$ is also important. As mentioned before, it records the restrictions on the quantified type variables in the expressions inside the branch. Let us explain what could go wrong if we did not impose the condition $\Delta \vdash \mathcal{L}_1 \sqsubseteq \mathcal{L}_2$ in the typecase checking rule.

Let $\mathcal{L}_1 = \{\ell_{\mathsf{int}}, \ell_{\mathsf{bool}}, \ell_\times\}$, and $\mathcal{L}_2 = \{\ell_{\mathsf{int}}, \ell_\times\}$. Then consider the following function:

$$\begin{array}{l} \mathsf{f} = \Lambda\alpha{:}\star\upharpoonright\mathcal{L}_1.\ \mathbf{typecase}\ \alpha\ \{ \\ \qquad \ell_{\mathsf{int}} \quad \Rightarrow \quad \lambda x{:}\mathsf{int}.\ \mathbf{true}, \\ \qquad \ell_{\mathsf{bool}} \quad \Rightarrow \quad \lambda x{:}\mathsf{bool}.\ \mathbf{false}, \\ \qquad \ell_\times \quad \Rightarrow \quad \Lambda\alpha_1{:}\star\upharpoonright\mathcal{L}_2.\ \Lambda\alpha_2{:}\star\upharpoonright\mathcal{L}_2. \\ \qquad\qquad\qquad \lambda x : \alpha_1 \times \alpha_2.\ \mathbf{typecase}\ \alpha_2\ \{ \\ \qquad\qquad\qquad\quad \ell_{\mathsf{int}} \Rightarrow \ldots, \\ \qquad\qquad\qquad\quad \ell_\times \Rightarrow \ldots\} \\ \quad \} \end{array}$$

This function will be well typed if we omit the restriction that $\vdash \mathcal{L}_1 \sqsubseteq \mathcal{L}_2$. However, when we call it as $\mathsf{f}\ [\mathsf{int} \times \mathsf{bool}]\ (2, \mathbf{true})\ (3, \mathbf{false})$ evaluation will become stuck as the inner typecase may only analyze integers and products. Therefore we require that the label set component of map types that specifies the restriction for polykinded types is always at least as big as the domain of the branch.

3.3 Properties

The $\lambda_{\mathcal{L}}$ language is type sound, following from the usual progress and preservation theorems [36]. The proofs of these theorems are inductions over the relations defined.

THEOREM 3.1 (PROGRESS). *If* $\ell_{\mathsf{int}}, \ell_{\to} \notin \mathrm{dom}(\Sigma)$ *and* $\vdash e : \tau \mid \Sigma$ *then either* e *is value or there exist some* $\mathcal{L}'$, e' *such that* $\mathcal{L}; e \mapsto \mathcal{L}'; e'$*, where* $\mathcal{L} = \mathrm{dom}(\Sigma) \cup \{\ell_{\mathsf{int}}, \ell_{\to}\}$.

THEOREM 3.2 (PRESERVATION). *If* $\ell_{\mathsf{int}}, \ell_{\to} \notin \mathrm{dom}(\Sigma)$, $\vdash e : \tau \mid \Sigma$ *and* $\mathcal{L}; e \mapsto \mathcal{L}'; e'$*, where* $\mathcal{L} = \mathrm{dom}(\Sigma) \cup \{\ell_{\mathsf{int}}, \ell_{\to}\}$, *then there exists* Σ'*, such that* $\mathcal{L}' = \mathrm{dom}(\Sigma') \cup \{\ell_{\mathsf{int}}, \ell_{\to}\}$, $\vdash e' : \tau \mid \Sigma'$ *and* $\Sigma \subseteq \Sigma'$.

We have also shown that the coercions are not necessary to the operational semantics [33]. A calculus where the coercions have been erased has the same operational behavior as this calculus. In other words, expressions in $\lambda_{\mathcal{L}}$ evaluate to a value if and only if their coercion-erased versions evaluate to the coercion-erased value. This justifies our claim that coercions have no operational cost.

4. FULL REFLEXIVITY

Full reflexivity [31] refers to the ability to analyze every type. It is arguably an important property for modern type-analyzing programming languages. For example distributed programming requires transferring data and code of arbitrary types through a communication channel. This data may be existential packages or the code may be consist of

$$
\begin{array}{lrl}
\textit{Kinds} & \kappa ::= & \chi \mid \star \mid \kappa_1 \to \kappa_2 \\
 & \mid & \mathrm{L}(\kappa_1) \to \kappa_2 \mid \mathrm{Ls} \to \kappa \mid \forall\chi.\kappa \\
\textit{Labels} & l ::= & \ldots \\
\textit{Label sets} & \mathcal{L} ::= & \ldots \\
\textit{Types} & \tau ::= & \alpha \mid l \mid \lambda\alpha{:}\kappa.\tau \mid \tau_1\ \tau_2 \\
 & \mid & \lambda\iota{:}\mathrm{L}(\kappa).\tau \mid \tau\ \hat{l} \\
 & \mid & \lambda s{:}\mathrm{Ls}.\tau \mid \tau\ \mathcal{L} \mid \Lambda\chi.\tau \mid \tau\ [\kappa] \mid \tau'\langle \tau : \kappa \restriction \mathcal{L} \rangle \\
\textit{Terms} & e ::= & \ldots \\
 & \mid & \textbf{setcase}\ \mathcal{L}\ \theta \\
 & \mid & \textbf{lindex}\ l \\
 & \mid & \Lambda\chi.e \mid e[\kappa] \\
\textit{Setcase branches} & \theta ::= & \{\varnothing \Rightarrow e_{\varnothing},\ \{\} \Rightarrow e_{\{\}}, \cup \Rightarrow e_{\cup},\ \mathcal{U} \Rightarrow e_{\mathcal{U}}\ \}
\end{array}
$$

Figure 5: Modifications for full reflexivity

ℓ_{int}	$: \star$	*integers*
$\ell_{\to}$	$: \star \to \star \to \star$	*function type creator*
$\ell_{\forall}$	$: \forall\chi.(\chi \to \star) \to \mathrm{Ls} \to \star$	*type polymorphism*
$\ell_{\forall^*}$	$: \forall\chi.(\mathrm{L}(\chi) \to \star) \to \star$	*label polymorphism*
$\ell_{\forall\#}$	$: (\mathrm{Ls} \to \star) \to \star$	*label set polymorphism*
$\ell_{\forall+}$	$: (\forall\chi.\star) \to \star$	*kind polymorphism*
ℓ_{map}	$: \mathrm{Ls} \to (\star \to \star) \to \mathrm{Ls} \to \star$	*map type*

Figure 6: Distinguished label kinds

polymorphic functions. Unless the language supports full reflexivity, and in particular, analysis of existential and universal types, generic communication of this data or code is impossible.

In core $\lambda_{\mathcal{L}}$, universal types and map types cannot be analyzed by **typecase**. In this section we present the full $\lambda_{\mathcal{L}}$ language, which addresses this shortcoming by extending the set of analyzed types to include all types.

The full language, shown in Figure 5, introduces distinguished labels to represent constructors of universal types and map types. The kinds of the distinguished labels are shown in Figure 6. These types now become syntactic sugar for applications of the appropriate labels, as shown in Figure 8. These new distinguished labels require new forms of abstractions at the type level; for labels ($\lambda\iota{:}\mathrm{L}(\kappa).\tau$), for label sets ($\lambda s{:}\mathrm{Ls}.\tau$) and for kinds ($\Lambda\chi.\tau$). The full language of kinds include the kinds of the core $\lambda_{\mathcal{L}}$, kinds for label abstractions ($\mathrm{L}(\kappa_1) \to \kappa_2$), kinds for label set constructors ($\mathrm{Ls} \to \kappa$) and universal kinds ($\forall\chi.\kappa$), which are the kinds of kind abstractions at the type level.

There is one implication to the addition of these new abstraction forms. Polykinded types cannot be determined statically in general, as the kind over which they are parameterized may be unknown at compile time. Therefore polykinded types are part of the syntax of the full language, instead of being derived forms, and the type equivalence relation includes the unfolding of polykinded types. The interesting equivalences are given in Figure 7. At run time, closed polykinded types will always reduce to one of the other type forms.

Since labels and label sets can be abstract as well, although not required for full reflexivity, it makes sense to

$$
\begin{array}{lcl}
\tau'\langle \tau : \star \restriction \mathcal{L} \rangle & \triangleq & \tau'\tau \\
\tau'\langle \tau : \kappa_1 \to \kappa_2) \restriction \mathcal{L} \rangle & \triangleq & \forall\alpha{:}\kappa\restriction\mathcal{L}.\tau'\langle \tau\ \alpha : \kappa_2 \restriction \mathcal{L} \rangle \\
\tau'\langle \tau : \mathrm{L}(\kappa_1) \to \kappa_2 \restriction \mathcal{L} \rangle & \triangleq & \forall\iota{:}\mathrm{L}(\kappa_1).\tau'\langle \tau\ \hat{\iota} : \kappa_2 \restriction \mathcal{L} \rangle \\
\tau'\langle \tau : \mathrm{Ls} \to \kappa \restriction \mathcal{L} \rangle & \triangleq & \forall s{:}\mathrm{Ls}.\tau'\langle \tau\ s : \kappa \restriction \mathcal{L} \rangle \\
\tau'\langle \tau : \forall\chi.\kappa \restriction \mathcal{L} \rangle & \triangleq & \forall\chi.\tau'\langle \tau\ [\chi] : \kappa \restriction \mathcal{L} \rangle
\end{array}
$$

Figure 7: Polykinded type equivalences

$$
\begin{array}{rlrl}
\mathsf{int} \triangleq & \ell_{\mathsf{int}} & \tau_1 \to \tau_2 \triangleq & \ell_{\to}\ \tau_1\ \tau_2 \\
\forall\alpha{:}\kappa\restriction\mathcal{L}.\tau \triangleq & \ell_{\forall}\ [\kappa]\ (\lambda\alpha{:}\kappa.\tau)\ \mathcal{L} & \forall\chi.\tau \triangleq & \ell_{\forall+}\ (\Lambda\chi.\tau) \\
\forall s.\tau \triangleq & \ell_{\forall\#}\ (\lambda s{:}\mathrm{Ls}.\tau) & \mathcal{L} \Rightarrow \tau \restriction \mathcal{L}' \triangleq & \ell_{\mathsf{map}}\ \mathcal{L}\ \tau\ \mathcal{L}' \\
\forall\iota{:}\mathrm{L}(\kappa).\tau \triangleq & \ell_{\forall^*}\ [\kappa]\ (\lambda\iota{:}\mathrm{L}(\kappa).\tau) & &
\end{array}
$$

Figure 8: Syntactic sugar for types

allow for label and label set analysis. To allow the programmer to learn about new labels, full $\lambda_{\mathcal{L}}$ introduces an operator **lindex** that returns the integer associated with its argument label constant. This operator provides the programmer a way to distinguish between labels at run time. The rule for **lindex** is straightforward.

$$\frac{}{\mathcal{L}; \textbf{lindex}\ \ell_i^{\kappa} \mapsto \mathcal{L}; i}$$

Another addition is that of a label set analysis operator **setcase**. The operator **setcase** has branches for all possible forms of label set—empty, singleton, union and universe. Operationally, **setcase** behaves much like **typecase**, converting its argument to a normal form, so that equivalent label sets have the same behavior, and then stepping to the appropriate branch.

To demonstrate label and label set analysis, consider the following example, a function that computes a string representation of any label set. Assume also that the language is extended with strings and operations for concatenation and conversion to/from integers.

```
fix settostring:$\forall s{:}\mathrm{Ls}.$ string.
  $\Lambda s{:}\mathrm{Ls}.$ setcase $s$
    {  $\varnothing$  $\Rightarrow$  "",
       $\cup$  $\Rightarrow$  $\Lambda s_1{:}\mathrm{Ls}.\ \Lambda s_2{:}\mathrm{Ls}.$
             (settostring[$s_1$]) ++
             " " ++(settostring[$s_2$]),
       {}  $\Rightarrow$  $\Lambda\chi.\ \Lambda\iota{:}\mathrm{L}(\chi).$ int2string(lindex($\iota$)),
       $\mathcal{U}$  $\Rightarrow$  "U"
    }
```

The **setcase** operator has a fixed number of branches, one for every form of label set. In the example above, when the label set is empty we return the empty string. If the label set is a join of two other label sets, we abstract these label sets and call **settostring** recursively on each one. When the label set is a singleton set, we first abstract the kind of the set's label and then the label itself and then print the integer associated with the label. Finally, if the label set is $\mathcal{U}$ we simply return a string.

The rule to type check **setcase** is below.

$$\frac{\begin{array}{c}\Delta \vdash \tau' : \mathrm{Ls} \to \star \\ \Delta; \Gamma \vdash e_{\varnothing} : \tau'\varnothing \,|\, \Sigma \qquad \Delta; \Gamma \vdash e_{\{\}} : \forall\chi.\forall\iota{:}\mathrm{L}(\chi).\tau'\{\iota\} \,|\, \Sigma \\ \Delta; \Gamma \vdash e_{\cup} : \forall s_1{:}\mathrm{Ls}.\forall s_2{:}\mathrm{Ls}.\tau'(s_1 \cup s_2) \,|\, \Sigma \\ \Delta; \Gamma \vdash e_{\mathcal{U}} : \tau'\mathcal{U} \,|\, \Sigma \qquad \Delta \vdash \mathcal{L} : \mathrm{Ls}\end{array}}{\begin{array}{l}\Delta\Gamma \vdash \textbf{setcase}\ \mathcal{L}\ \ \{\varnothing \Rightarrow e_{\varnothing},\ \{\} \Rightarrow e_{\{\}}, \\ \qquad\qquad\qquad\quad \cup \Rightarrow e_{\cup},\ \mathcal{U} \Rightarrow e_{\mathcal{U}}\ \} : \tau'\ \mathcal{L} \mid \Sigma\end{array}}$$

In this rule, $e_{\{\}}$ must be able to take any label as its argument, whatever the kind of the label. Therefore, $\lambda_{\mathcal{L}}$ must support kind polymorphism, as shown in Figure 5.

The operational semantics for **setcase** expressions resembles the semantics of **typecase**. The most interesting rules are the one for singleton sets and joins of label sets.

$$\frac{\mathcal{L} \downarrow_* \{\ell_i^\kappa\} \qquad \theta = \{\varnothing \Rightarrow e_{\varnothing},\ \{\} \Rightarrow e_{\{\}}, \cup \Rightarrow e_{\cup},\ \mathcal{U} \Rightarrow e_{\mathcal{U}}\ \}}{\mathcal{L}; \textbf{setcase}\ \mathcal{L}\ \theta \mapsto \mathcal{L}; e_{\{\}}[\kappa][\widehat{\ell_i^\kappa}]}$$

$$\frac{\begin{array}{c}\mathcal{L} \downarrow_* \mathcal{L}_1 \cup \mathcal{L}_2 \\ \theta = \{\varnothing \Rightarrow e_{\varnothing},\ \{\} \Rightarrow e_{\{\}}, \cup \Rightarrow e_{\cup},\ \mathcal{U} \Rightarrow e_{\mathcal{U}}\ \} \\ \mathcal{L}_1 \cup \mathcal{L}_2\ \textit{norm}\end{array}}{\mathcal{L}; \textbf{setcase}\ \mathcal{L}\ \theta \mapsto \mathcal{L}; e_{\cup}[\mathcal{L}_1][\mathcal{L}_2]}$$

If the label set is reduced to a singleton set, we apply the appropriate branch expression to the kind and the label of the set. The label set reduction relation $\mathcal{L}_1 \downarrow_* \mathcal{L}_2$ reorders the labels in a label set and eliminates empty components or reduces the label set to $\mathcal{U}$ if it contains $\mathcal{U}$. The relation $\mathcal{L}_1 \cup \mathcal{L}_2$ *norm* "splits" a label set in a canonical way, namely $\mathcal{L}_1$ will be a singleton set that contains the label with the smallest index in the label set. If the label set is reduced to a join of two other label sets, we apply the appropriate branch expression to these two label sets. The rest of the evaluation rules, as well as the semantics of label set reductions, are omitted for reasons of space but can be found in the technical report [33].

5. OTHER EXTENSIONS

A difficulty of working with $\lambda_{\mathcal{L}}$ is that **typecase** must always have a branch for the label of its argument. However it is natural to provide *default branches* that apply when no other branches match a label. Default branches could be then combined with linguistic mechanisms, such as exceptions, to provide yet another way to treat new names.

Another practical extension is the addition of *records and variants*. Our approach is to use the existing label mechanism to manipulate the names of fields of records or variants. Variant and record types are formed by applying their corresponding constructors to finite first-class maps from labels to types. The result is system that resembles row polymorphism [26].

In the case of recursive types, higher order coercions cannot completely eliminate a new label from the type of an expression. The coercion only unrolls the type, leaving an occurrence of a new label. The addition of a *recursive uncoercion* mechanism would then improve significantly the practicality of the language.

We have worked through the details of these extensions and added record and variant support, as well as hooks for default branches in the implementation of the full language. We omit the details for reasons of space.

6. RELATED WORK

There is much research on type-directed programming. *Run-time type analysis* allows the structural analysis of dynamic type information. Abadi, et al. introduced a type-dynamic to which types could be coerced, and later via case analysis, extracted [1]. The core semantics of **typecase** in $\lambda_{\mathcal{L}}$ is similar to the intensional polymorphism of Harper and Morrisett [13]. However, $\lambda_{\mathcal{L}}$ does not include a type-level analysis operator. Our extension of $\lambda_{\mathcal{L}}$ to be fully reflexive follows a similar extension of Harper and Morrisett's language by Trifonov, Saha, and Shao [31]. Weirich [35] extended run-time analysis to higher-order type constructors following the work of Hinze [14].

In intensional type analysis, type information is passed at run time. This results in a duplication of the operational semantics of the language for types and terms. Moreover, since types can be analyzed we lose parametricity and type abstraction. To solve these problems, among others, Crary, Weirich and Morrisett proposed an erasure semantics for intensional type analysis [5], where special terms that serve as type representations are constructed and passed at run time. Type abstraction is preserved by making the term representation of certain types unavailable at run time. Using an erasure semantics would not be simple in our case. We would have to use idioms like row polymorphism [27] and polymorphic variants [8] to represent extensible term-level representations of types. However, using polymorphic variants off-the-shelf does not work. The reason is that expressions in typecase branches do not have necessarily the same return types. For example, in a polymorphic equality function each branch returns a term of different type. There is no straightforward way to encode this fact using term-based pattern matching and extensible variants. Finally, even though we do not have an erasure semantics we can still enforce some abstraction through new types and unanalyzable types.

Generic programming uses the structure of datatypes to generate specialized operations at compile time. The Charity language [3] automatically generates folds for datatypes. PolyP [17] is an extension of Haskell that allows the definition of polytypic operations based on positive, regular datatypes. Functorial ML [19] bases polytypic operations on the composition of functors, and has lead to the programming language FISh [18]. Generic Haskell [2], following the work of Hinze [14] allows polytypic functions to be indexed by any type or type constructor.

Nominal forms of ad-hoc polymorphism are usually used for *overloading*. Type classes in Haskell [34] implement overloading by defining classes of types that have instances for a set of polytypic operations. Hinze and Peyton Jones [15] explored an extension to automatically derive type class instances by looking at the underlying structure of new types. Dependency-style Generic Haskell [22] revises the Generic Haskell language to be based on the names of types instead of their structure. However, to automatically define more generic functions, it converts user-defined types into their underlying structural representations if a specific definition has not been provided.

Many languages use a form of generative types to represent application-specific abstractions. For example, Standard ML [23] and Haskell [25] rely on datatype generativity in type inference. Modern module systems also provide generative types [6]. When the definition of the new type is known, the type isomorphisms of this paper differ from calculi with type equalities (such as provided by Harper and Lillibridge [12] or Stone and Harper [30]) in that they require explicit terms to coerce between a type name and its definition. While explicit coercions are more difficult for the programmer to use, they simplify the semantics of the generative types.

A few researchers have considered the combination of generative types with forms of dynamic type analysis. Glew's [10] source language dynamically checks pre-declared subtyping relationships between type names. Lämmel and Peyton Jones [20] used dynamic type equality checks to implement a number of polytypic iterators. Rossberg's λ_{N} calculus [29] dynamically checks types (possibly containing new names) for equality. Rossberg's language also includes higher-order coercions to allow type isomorphisms to behave like existentials, hiding type information inside a precomputed expression. However, his coercions have a different semantics from ours. Higher-order coercions are reminiscent of the colored brackets of Grossman et al. [11], which are also used by Leifer et al. [21] to preserve type generativity when marshaling.

7. DISCUSSION

In conclusion, the $\lambda_{\mathcal{L}}$ language provides a framework that can model both open and closed type analysis in a nominal setting. Because it can represent both forms, it makes apparent the advantages and disadvantages of each. We view $\lambda_{\mathcal{L}}$ as a solid foundation for the design of a user-level language that incorporates both versions of polytypism.

There are certain drawbacks in $\lambda_{\mathcal{L}}$. Although the flexibility of having unanalyzable types—by using $\mathcal{U}$ in type abstractions—is important, this is not the best way to do it. It does not allow types to be partly abstract and partly transparent.

Apart from developing a usable source language, there are a number of other extensions that would be worthwhile to consider. First, our type definitions provide a simplistic form of generativity; we plan to extend $\lambda_{\mathcal{L}}$ with a module system as a more realistic model. Furthermore, type analysis is especially useful for applications such as marshaling and dynamic loading, so it would be useful to develop a distributed calculus based on $\lambda_{\mathcal{L}}$. To avoid the need for a centralized server to provide unique type names, name generation could be done randomly from some large domain with very low probability of collision.

We plan to extend $\lambda_{\mathcal{L}}$ in two ways to increase its expressiveness. First, **typecase** makes restrictions on all labels that appear in its argument so that it can express catamorphisms over the structure of the type language. However, not every type-directed function is a catamorphism. Some operations only determine the head form of the type. Others are hybrids, applicable to a specific pattern of type structure. For example, if we were to add references to the calculus, we could extend **eq** to all references, even if their contents are not comparable, by using pointer equality. This calculus cannot express that pattern. Furthermore, some operations are only applicable to very specific patterns. For example, an operation may be applicable only to functions that take integers as arguments, such as functions of the form $\mathsf{int} \rightarrow \mathsf{int}$ or $\mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{int}$. These operations are still expressible in the core calculus, but there is no way to statically determine whether the type argument satisfies one of these patterns, so dynamic checks must be used. To approach this problem, we plan to investigate pattern calculi that may be able to more precisely specify the domain of type-directed operations. For example, the mechanisms of languages designed to support native XML processing [16, 7] can statically enforce that tree-structured data has a very particular form.

Second, it is also important to add type-level analysis of types. As shown in past work [13, 31], it is impossible to assign types to some type-directed functions without this feature. One way to do so might be to extend the primitive-recursive operator of Trifonov et al. [31] to include first-class maps from labels to types.

Acknowledgments

Thanks to Steve Zdancewic, Benjamin Pierce, and Andreas Rossberg for helpful discussion. We also benefitted greatly from insightful comments by anonymous reviewers on earlier revisions of this work.

8. REFERENCES

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[2] D. Clarke, R. Hinze, J. Jeuring, A. Löh, and J. de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Utrecht University, 2001.

[3] R. Cockett and T. Fukushima. About Charity. Yellow Series Report No. 92/480/18, Department of Computer Science, The University of Calgary, June 1992.

[4] K. Crary and S. Weirich. Flexible type analysis. In *Proceedings of the Fourth International Conference on Functional Programming (ICFP)*, pages 233–248, Paris, Sept. 1999.

[5] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, Nov. 2002.

[6] D. Dreyer, K. Crary, and R. Harper. A type system for higher-order modules. In *Proceedings of the Thirtieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 236–249. ACM Press, 2003.

[7] V. Gapeyev and B. Pierce. Regular object types. In *Proc. 10th International Workshops on Foundations of Object-Oriented Languages, FOOL '03*, New Orleans, LA, USA, Jan. 2003.

[8] J. Garrigue. Programming with polymorphic variants. In *ML Workshop*, Sept. 1998.

[9] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[10] N. Glew. Type dispatch for named hierarchical types. In *1999 ACM SIGPLAN International Conference on Functional Programming*, pages 172–182, Paris, France, Sept. 1999.

[11] D. Grossman, G. Morrisett, and S. Zdancewic. Syntactic type abstraction. *ACM Transactions on Programming Languages and Systems*, 22(6):1037–1080, Nov. 2000.

[12] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-First ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 123–137, Portland, Oregon, Jan. 1994.

[13] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, Jan. 1995.

[14] R. Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2–3):129–159, 2002. MPC Special Issue.

[15] R. Hinze and S. Peyton Jones. Derivable type classes. In G. Hutton, editor, *Proceedings of the Fourth Haskell Workshop, Montreal, Canada, September 17, 2000*, volume 41.1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, Aug. 2000.

[16] H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. *Journal of Functional Programming*, 13(4), 2002.

[17] P. Jansson and J. Jeuring. PolyP—A polytypic programming language extension. In *Twenty-Fourth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482, Paris, France, 1997.

[18] C. Jay. Programming in FISh. *International Journal on Software Tools for Technology Transfer*, 2:307–315, 1999.

[19] C. B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, Nov. 1998.

[20] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)*, 2003.

[21] J. J. Leifer, G. Peskine, P. Sewell, and K. Wansbrough. Global abstraction-safe marshalling with hash types. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 87–98, Uppsala, Sweden, 2003.

[22] A. Löh, D. Clarke, and J. Jeuring. Dependency-style generic haskell. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 141–152, 2003.

[23] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Massachusetts, 1997.

[24] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *FPCA95: Conference on Functional Programming Languages and Computer Architecture*, pages 66–77, La Jolla, CA, June 1995.

[25] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[26] D. Rémy. Records and variants as a natural extension of ML. In *Sixteenth Annual Symposium on Principles Of Programming Languages*, 1989. Available from `http://doi.acm.org/10.1145/75277.75284`.

[27] D. Rémy. Type inference for records in a natural extension of ML. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[28] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, 1983. Proceedings of the IFIP 9th World Computer Congress.

[29] A. Rossberg. Generativity and dynamic opacity for abstract types. In D. Miller, editor, *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, Uppsala, Sweden, Aug. 2003. ACM Press.

[30] C. Stone and R. Harper. Deciding type equivalence in a language with singleton kinds. In *Twenty-Seventh ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 214–225, Boston, MA, USA, Jan. 2000.

[31] V. Trifonov, B. Saha, and Z. Shao. Fully reflexive intensional type analysis. In *Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 82–93, Montreal, Sept. 2000.

[32] J. C. Vanderwaart, D. Dreyer, L. Petersen, K. Crary, R. Harper, and P. Cheng. Typed compilation of recursive datatypes. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, New Orleans, USA, Jan. 2003.

[33] D. Vytiniotis, G. Washburn, and S. Weirich. An open and shut typecase. Technical Report MS-CIS-04-26, University of Pennsylvania, 2004. Available from `http://www.cis.upenn.edu/~dimitriv/itaname`.

[34] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Sixteenth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 60–76. ACM Press, 1989.

[35] S. Weirich. Higher-order intensional type analysis. In D. L. Métayer, editor, *11th European Symposium on Programming*, pages 98–114, Grenoble, France, 2002.

[36] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

[37] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. Technical Report IC/2004/33, EPFL Lausanne, Switzerland, 2004.

`$Id: itaname.tex 332 2004-09-16 22:39:46 -0400 dimitriv $`