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

Ad-hoc polymorphism allows the execution of programs to depend on type information. It is a compelling addition to typed programming languages. For example, it may be used to implement generic operations over data structures, such as equality, marshalling, and iteration.

There are two different forms of ad-hoc polymorphism. With the nominal form, the execution of an operation is determined solely by the name of the type argument, whereas with the structural form operations decompose the structure of the type. These two forms of ad-hoc polymorphism differ in the way that they treat user-defined types. Operations defined by the nominal approach are "open"—they must be extended with specialized branches for user-defined types. However, t may be tedious and even difficult to add new operations that apply to many types. In contrast, structurally defined operations are closed to extension. They automatically apply to user-defined types by treating them as equal to their underlying definitions. This approach destroys the distinctions that user-defined types are designed to express.

Both approaches have their benefits, so it important to provide both capabilities in a single language. Therefore we present an expressive language that supports both forms of ad-hoc polymorphism.

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*

1. INTRODUCTION

With ad-hoc polymorphism the execution of programs depends on type information. A parametrically polymorphic function must behave the same for all instantiations. However, the instance of an ad-hoc polymorphic function for integers may behave differently from the instance for booleans. We call functions that depend on type information *type-directed*.

Ad-hoc polymorphism is a compelling addition to a typed programming language. It is well suited for dynamic environments where it can be used to implement dynamic typing, dynamic loading and marshalling. It is also essential to the definition of generic versions of many basic operations such as equality and structural traversals. In particular, ad-hoc polymorphism simplifies programming with complicated data structures, eliminating the need for repetitive "boilerplate code". For example, the implementation of a compiler may include many data structures for representing intermediate languages and many passes over these data structures. Without type-directed programming, the same code for traversing abstract syntax must be implemented for each intermediate language. The generic traversals defined by ad-hoc polymorphism allow the programmer to concentrate on the important parts of a transformation.

Currently, there are two forms of ad-hoc polymorphism in typed, functional languages. The first is based on the *nominal* analysis of type information, such as Haskell type classes [30]. The execution of an ad-hoc operation is determined solely by the name of the type argument (or the name of its head constructor, such as list.)

For example, we may use type to implement a polymorphic structural equality. The type class declares that there is a type-directed operation called `eq` and each instance of the class describes how `eq` behaves for that type. For composite types, such as products and lists, equality is defined in terms of equality for the components of the type.

```
class Eq a where
  eq :: a -> a -> Bool
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 y1 && eq x2 y2
instance (Eq a) => Eq [a] where
  eq l1 l2 = all2 eq l1 l2
```

Nominal analysis naturally limits the domain of an ad-hoc operation to those types where a definition has been provided. For example, `eq` is not defined for function types. Type-directed operations in a nominal framework are naturally "open"; at any time they may be extended with in-

ICFP'04, September 19–22, 2004, Snowbird, Utah.

stances for new types, without modifying existing code.

The second form of ad-hoc polymorphism is based on the *structural* analysis of types. For example, intensional type analysis [11] allows programmers to define type-directed operations by case analysis of the type structure. Polymorphic equality defined by run-time type analysis may look like the following:

```
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) && (snd x)(snd y)
     [b]    -> all2 eq x y
     (b->c) -> error "eq not defined for functions"
```

Because type-directed operations are defined by case analysis, they are naturally "closed" to extension. In fact, cases for all types must be provided when such operations are defined.

These two forms of ad-hoc polymorphism differ in the way that they treat user-defined types. User-defined types such as Haskell's `newtypes` [23], are an important part of many languages. Although these new types are isomorphic to existing types, they express application-specific distinctions that can be made by the type checker. For example, a programmer may wish to ensure that he does not confuse phone numbers with ages in an application, even though both may be represented using integers.

Because nominal operations are open, they must be extended with instances for each new user-defined type. It may be tedious and even difficult to add new operations that apply to many types. Furthermore, if some types are defined in separate inaccessible modules then it is impossible for the programmer to extend the operation to those types. Instead, he must rely on the definer of the type to add the instance, and there is no guarantee that she will respect the invariants of the type-directed operation.

On the other hand, closed operations cannot be extended to new types. Structural systems treat new types as equal to their definitions. This approach destroys the distinctions that the new types are designed to express. A type-directed operation cannot treat an age differently from a phone number—both are treated as integers. While some systems allow ad-hoc definitions for user-defined types, there is a loss of abstraction—a type-directed operation can always determine a type's underlying representation.

In the presence of user-defined types, neither purely nominal nor purely structural ad-hoc polymorphism is entirely satisfactory.

1.1 Combining both forms in one language

This paper unifies the two different forms of ad-hoc polymorphism in a foundational language, called $\lambda_{\mathcal{L}}$. This language provides capabilities for both structural and nominal analysis in a coherent framework, allowing developers to choose which characteristics they wish to use from each system.

At the core, $\lambda_{\mathcal{L}}$ is a simple system for structural type analysis augmented with user-defined types. The structural analysis operator **typecase** may include branches for these new names if they are in scope. Naturally, some type-directed operations may be unable to handle some newly defined types. Types containing names for which there is no branch in an operation cannot be allowed as an argument, or evaluation will become stuck. Therefore, the type system of $\lambda_{\mathcal{L}}$ statically tracks the names used in types and compares them to the domain of a type analysis operation.

New names are generated dynamically during execution, so it is desirable to extend type-directed operations with branches for these new names. For this purpose, we introduce first-class maps from names to expressions. Intuitively, these maps are branches for **typecase** that may be passed to type-directed operations, extending them to handle the new names. Also, $\lambda_{\mathcal{L}}$ includes support to coerce the types of expressions so that they do not mention new type names.

We stress that we do not consider $\lambda_{\mathcal{L}}$ an appropriate source language for humans, in much the same way that F_ω is not an appropriate source language for humans. As defined, $\lambda_{\mathcal{L}}$ requires that programs be heavily annotated and written in a highly-stylized fashion. The next step in this research program is to develop automated assistance for the common idioms, such as the inference of type arguments and first-class maps.

1.2 Contributions of this work

The $\lambda_{\mathcal{L}}$ language is an important step towards improving the practicality of type-directed programming. In particular, this paper has the following contributions:

- We define a language that allows the definition of both "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 [11, 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 [31] 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 [23, 29, 26] to higher-order coercions in the presence of type constructor isomorphisms.

The remainder of this paper is as follows. In the next section we introduce the features of $\lambda_{\mathcal{L}}$ through examples. We first describe the semantics of the core language in Section 3, and then extend it to be fully reflexive in Section 4. In Section 5 we show how higher-order analysis may be defined and discuss additional extensions in Section 6. We discuss related work in Section 7 and conclude in Section 8.

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

At the core, $\lambda_{\mathcal{L}}$ is a polymorphic lambda calculus (F_ω) [7, 25], augmented with type analysis and user-defined types. The syntax of $\lambda_{\mathcal{L}}$ appears in Figure 1. In addition to the standard kinds, type constructors and terms of F_ω, $\lambda_{\mathcal{L}}$ includes labels (l) and sets of labels ($\mathcal{L}$). Labels may be considered to be "type constants" and model both built-in types (such as `int`) and user-defined types.

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

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

An important point is that $\lambda_{\mathcal{L}}$ supports *run-time* analysis of type information instead of requiring that all type-directed operations be resolved at compile time. Run-time analysis is necessary because there are many situations where types are not known at compile time. For example, large programs, where the benefit of type-directed programming is most important, are not compiled in their entirety. Furthermore, separate compilation, dynamic loading or run-time code generation requires run-time type analysis. Even within a single compilation unit, not all type information may be available at compile time because of first-class polymorphism (where a data structure may hide some type information) or polymorphic recursion (where each iteration of a loop is instantiated with a different type).

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

2.1 Generative types

The $\lambda_{\mathcal{L}}$ language includes a simple mechanism for users to define new type constants. We call all type constants *labels* to emphasize the fact that they do not α-vary. Arbitrary label constants are written as ℓ^{κ}_i, annotated with their kinds. Some distinguished constants in this language are constructors for primitive types. The label $\ell^{\star}_0$ is a nullary constructor for the type of integers, and $\ell^{\star\rightarrow\star\rightarrow\star}_1$ 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 those types.

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 $\{\!\{\cdot\}\!\}^{-}_{\iota=\tau}$ 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.

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

Unlike other forms of user-defined types, such as Haskell `newtype`s, this mechanism dynamically creates new "types". Generating these new labels requires an operational effect at run time. However, the coercions that convert between the new label and its definition have no run-time cost. We chose this mechanism to model generative types in $\lambda_{\mathcal{L}}$ for its simplicity. A more sophisticated language could base its mechanism for type generativity on a module system.

Note that even though run-time type analysis destroys the parametricity created by a type polymorphism ($\Lambda\alpha$) users may still hide the implementation details of abstract datatypes with these generative types. Once outside the scope of a new label, it is impossible to determine its underlying definition. For example, we know that the polymorphic function f below must treat its term argument parametrically because, even in the presence of run-time type analysis, it cannot coerce it to the type 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 $\mathbf{typecase}\ \tau\ e$ may be used to define type-directed operations in $\lambda_{\mathcal{L}}$. This operator determines the head (or outermost) 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 $\mathbf{typecase}$ may be formed from 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 second map has precedence. 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 matching branch for the analyzed type. For example, stuck expressions such as $\mathbf{typecase}\ \mathsf{bool}\ \{\ell_{\mathsf{int}} \Rightarrow 2\}$ should not type check, because there is no branch for the boolean type (or rather its label).

For this reason, when checking a $\mathbf{typecase}$ 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 have branches in $\mathbf{typecase}$. Label sets in $\lambda_{\mathcal{L}}$ may be 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, $\mathcal{U}$. Analogously to finite maps, $\{l_1, \ldots, l_n\}$ abbreviates $\{l_1\} \cup \ldots \cup \{l_n\}$.

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 will be instantiated only by 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 $\mathcal{U}$ then it is unanalyzable because no **typecase** can cover all branches.[1]

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

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

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 **typecase** is determined by the kind of the matched label. After **typecase** determines the head label of its argument, it steps to the corresponding map branch and applies that branch to any 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.

$$\begin{array}{l}
\textbf{eq}[\mathsf{list\ int}]\mapsto(\Lambda\beta{:}\star\restriction\mathcal{L}_0.\ \lambda x{:}(\mathsf{list}\ \beta).\ \lambda y{:}(\mathsf{list}\ \beta).\\
\qquad\qquad\qquad \textbf{all2}\ (\textbf{eq}[\beta])\ x\ y)\ \ [\mathsf{int}]\\
\qquad\quad \mapsto\lambda x{:}(\mathsf{list\ int}).\ \lambda y{:}(\mathsf{list\ int}).\ \textbf{all2}\ (\textbf{eq}[\mathsf{int}])\ x\ y
\end{array}$$

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 $\ell_{\rightarrow}$ from the set of labels for the argument of **eq**.

2.3 Generative types and type analysis

The function **eq** is closed to extension. However, with the creation of new labels there may be many more 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 application, coerce all the arguments to **eq** so that their types do not contain new labels.

2.3.1 *Extensible type analysis*

In $\lambda_{\mathcal{L}}$, we can rewrite **eq** to be extensible 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 $\mathcal{L}_1\Rightarrow\tau\restriction\mathcal{L}_2$. The first component of this type is the domain of the map. The second and third components are used to describe the types of these branches.[2] Using first-class maps, we can pass a branch for int's into the following operation:

$$\begin{array}{l}
\lambda x{:}(\{\ell_{\mathsf{int}}\}\Rightarrow(\lambda\alpha{:}\star.\mathsf{bool})\restriction\{\ell_{\mathsf{int}}\}).\\
\quad\textbf{typecase}\ \mathsf{int}\ (\{\ell_{\mathsf{bool}}\Rightarrow\textbf{true}\}\bowtie x)
\end{array}$$

For simplicity, $\lambda_{\mathcal{L}}$ makes no attempt to enforce that the domains of joined maps are disjoint. Instead, maps are ordered, and existing branches may be shadowed because the rightmost matching branch will be selected. In the following expression, if $\{\ell_{\mathsf{int}}\Rightarrow\textbf{false}\}$ is supplied for x, the expression will evaluate to **false**.

$$\begin{array}{l}
\lambda x{:}(\{\ell_{\mathsf{int}}\}\Rightarrow(\lambda\alpha{:}\star.\mathsf{bool})\restriction\{\ell_{\mathsf{int}}\}).\\
\quad\textbf{typecase}\ \mathsf{int}\ (\{\ell_{\mathsf{bool}}\Rightarrow\textbf{false},\ell_{\mathsf{int}}\Rightarrow\textbf{true}\}\bowtie x)
\end{array}$$

Redefining the behavior of **typecase** for int may not be what the programmer intended, but allowing such a scenario does not affect the soundness of $\lambda_{\mathcal{L}}$. If the programmer wished to prevent this redefinition, she could join x on the left.

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**. We call functions that have been defined in this manner "open". For example, we can create an open version of **eq** as follows (again let $\mathcal{L}_0=\{\ell_{\mathsf{int}},\ell_{\mathsf{bool}},\ell_{\times},\ell_{\mathsf{list}}\}$). In the code below, s is a variable describing the domain of labels in the map y. The **eq** function may be instantiated with types containing labels from $\mathcal{L}_0$ or s.

$$\begin{array}{l}
\Lambda s{:}\mathrm{Ls}.\lambda y{:}(s\Rightarrow(\lambda\alpha{:}\star.\alpha\rightarrow\alpha\rightarrow\mathsf{bool})\restriction s\cup\mathcal{L}_0).\\
\quad\textbf{fix eq}{:}\forall\alpha{:}\star\restriction(s\cup\mathcal{L}_0).\ \alpha\rightarrow\alpha\rightarrow\mathsf{bool}.\\
\qquad\Lambda\alpha{:}\star\restriction(s\cup\mathcal{L}_0).\ \textbf{typecase}\ \alpha\\
\qquad\quad y\bowtie\{\ell_{\mathsf{int}}\Rightarrow\ldots,\ell_{\mathsf{bool}}\Rightarrow\ldots,\ell_{\times}\Rightarrow\ldots,\ell_{\mathsf{list}}\Rightarrow\ldots\}
\end{array}$$

With this version of **eq**, we can treat labels differently from their underlying representations. For example, if dollar amounts are stored as floating point numbers, we can round them to two decimal places before comparing them.

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

[1] While the flexibility of having unanalyzable types is important, this approach is not the best way to support parametric polymorphism—it does not allow types to be partly abstract and partly transparent.

[2] The type of a branch is determined by the kind of the label matched by that branch, as well as these two components. The precise specification of that relationship appears in Section 3.

to **tostring**, as below.

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

Dependency-Style Generic Haskell [20] 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

Not all type-directed operations should be extensible. Programmers may wish to reason about their operation in a "closed world". Furthermore, even if an operation is extensible, for many new labels the behavior of the operation should be identical to that for their underlying representation. Although a data structure containing a new label ι may be converted so that it does not mention the new label (by repeated use of $\{\!\!\{\cdot\}\!\!\}^-_\iota$), it still may be difficult or computationally expensive to coerce the components of a large data structure. For example, coercing a list of ι's to a list of int's requires deconstructing the list, coercing each element individually and creating a new list.

To avoid this unnecessary effort, both when the program is written and when it is executed, we add *higher-order coercions* to $\lambda_{\mathcal{L}}$. These terms provide an efficient mechanism for coercing values with labeled types between their underlying representations and back. Like first-order coercions, these operations have no run-time effect; they merely alter the types of expressions.

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

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

However, operationally, this map destructs and rebuilds the list, which could be computationally expensive. Higher-order coercions can coerce x to be of type $\mathsf{list\ int}$ without computational cost.

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

In general, a higher-order coercion is annotated with a type constructor (in this case list) that describes the location of the label to coerce in the type of the term.

Type Contexts	$\Delta ::= \cdot$		*empty context*
	$\mid$	$\Delta, \alpha{:}\kappa$	*λ-bound type vars*
	$\mid$	$\Delta, \iota{:}\mathrm{L}(\kappa)$	*label variables*
	$\mid$	$\Delta, s{:}\mathrm{Ls}$	*label set variables*
	$\mid$	$\Delta, \alpha{:}\kappa \upharpoonright \mathcal{L}$	*Λ-bound type vars*
Signatures	$\Sigma ::= \cdot \mid \Sigma, l{:}\kappa = \tau$		
Term Contexts	$\Gamma ::= \cdot \mid \Gamma, x{:}\tau$		
Values	$v ::= \lambda x{:}\sigma.e \mid i \mid \{\!\!\{v\}\!\!\}^+_{l=\tau}$		
	$\mid$	$\varnothing \mid \{l \Rightarrow e\} \mid v_1 \bowtie v_2$	
	$\mid$	$\Lambda\alpha{:}\kappa \upharpoonright \mathcal{L}.e \mid \Lambda\iota{:}\mathrm{L}(\kappa).e \mid \Lambda s{:}\mathrm{Ls}.e$	
Tycon paths	$\rho ::= \bullet \mid \rho\ \tau$		
Term paths	$p ::= \bullet \mid p\ [\tau]$		

Figure 2: Syntax necessary for the static and dynamic semantics

3. THE CORE LANGUAGE

Next we describe the semantics of core $\lambda_{\mathcal{L}}$ in detail, including the dynamic and static semantics of the mechanisms described in the previous section. The semantics of this language is defined by a number of judgments, the most important of which are described below. These judgments employ a number of new syntactic categories, which are listed in Figure 2. For reference, the complete semantics of this language appears in Appendix A.

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 described by Σ. Type isomorphisms are induced by **new** expressions that introduce new label variables isomorphic to types. 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. It says that a term e with a set of labels $\mathcal{L}$ steps to a new term e' with a possibly larger set of labels $\mathcal{L}'$. During the evaluation of the **new** operator, the label-set component allows the selection of a fresh label that has not previously been used. In this way, it resembles an *allocation semantics* [22, 8]. The initial state of execution includes all type constants, such as 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:

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

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

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

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

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

Figure 3: Operational semantics for higher-order coercions (excerpt)

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

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. 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. A path is a sequence of applications of several type arguments. Operationally, the primitive coercion "-" cancels the primitive coercion "+".

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

Higher-order coercions extend the expressiveness of the primitive coercions to allow the non-head positions of a type to change. As described in the last section, they are useful for coercing the types of values stored in data structures. These coercions are annotated with a type constructor τ' that describes the part of the data structure to be coerced.

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

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

Intuitively, a higher-order coercion "maps" the primitive coercions over an expression, guided by the type constructor τ'. Figure 3 lists some of the rules that describe the operational semantics of this term. The weak-head normal form of the constructor τ' determines the operation of higher-order coercions. This form is determined through the following kind-directed relation:

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

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

The first rule assures that if a type is of kind $\star$, then it normalizes to its weak-head normal form. The relation $\Delta \vdash \tau \Downarrow \tau'$ is a standard weak-head reduction relation, and is listed in the Appendix. 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.

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 a path headed by a variable or constant, a universal type, or a branch type. The form of τ determines the execution of the higher-order 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 (i.e., 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 value 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 the value is a function, then semantics pushes the coercion through the function, changing the type of its argument and the body of the function. Similar rules apply to other value forms.

3.2 Semantics of type analysis

The rule describing the execution of **typecase** is below:

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

This rule uses the relation $\Delta \vdash \tau \downarrow \tau'$ 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 some series of type arguments as specified by the term path p. This term path is derived from ρ in an obvious fashion.

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

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

The most important part of this rule is that it checks that τ may be safely analyzed by **typecase**. Whatever the head of the normal form of τ is, there must be a corresponding branch in **typecase**. 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 important rules for this judgment are those for labels and variables.

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

In the first rule above, labels are added to the set when they are used as types. The second two rules correspond to the two forms of type variable binding. Type variables bound from the term language are annotated with the set of labels that may appear in types that are used to instantiate them. However, variables that are bound by type-level abstractions do not have any such annotation, and consequently do not contribute to the label set. This last rule is sound because the appropriate labels will be recorded when the type-level abstraction is applied.

Not all types are analyzable in the core $\lambda_{\mathcal{L}}$ language. 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. In the next section, we show how to extend the calculus so that such types may be represented by labels, and therefore analyzed. For this core language however, we prevent such types from being the argument to **typecase** by not including rules to determine a label set for those types.

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 $\mathcal{L}$, this rule checks that the map can handle all possible labels in τ with $\mathcal{L} \sqsubseteq \mathcal{L}_1$.

The result type of **typecase** depends on the type of the map argument, $\mathcal{L}_1 \Rightarrow \tau \upharpoonright \mathcal{L}_2$. The most important rule for checking maps is the rule for singleton maps below.

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

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** will apply 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 each branch to any 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.

It is important for the expressiveness of this calculus that the **typecase** rule conservatively determines the set of labels that may occur *anywhere* in its type argument. It is also sound to define a version of this rule that determines the possible labels in the head position of the type, because that is all that are examined by **typecase**. However, in that case, branches that match labels of higher kinds must use $\mathcal{U}$ as the restriction for their quantified type variables. Only determining the head labels of types does not provide any information about the labels of other parts of the type.

That precision would prevent important examples from being expressible in this calculus. Many type-directed operations (such as polymorphic equality) are folds or catamorphisms over the structure of types. To determine the behavior of the algorithm for composite types, such as product types, the function must make recursive calls for the subcomponents of the type. Those recursive calls will type check only if we can show that the subcomponents satisfy the label set requirements of the entire operation. But as mentioned above, it must be assumed that those subcomponents have label set $\mathcal{U}$ and are unanalyzable.

3.3 Properties

The $\lambda_{\mathcal{L}}$ language is type sound, following from the usual progress and preservation theorems [32]. The proofs of these theorems are inductions over the derivations defined.

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

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

We have also shown that the coercions are not necessary to the operational semantics. An untyped calculus where the coercions have been erased (preserving types for analysis) 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.

4. FULL REFLEXIVITY

The core language demonstrates the basic idea of extensible **typecase** expressions, but does not offer the capability of full reflexivity. Some types cannot be analyzed by **typecase**. The full $\lambda_{\mathcal{L}}$ language addresses this problem and extends the set of analyzable types to include all types. For more expressiveness, the full language also includes label and label set runtime analysis operators. In the rest of this section we discuss these extensions. The modifications to

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

Figure 4: Modifications for full reflexivity

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

Figure 5: Distinguished label kinds

the syntax of core $\lambda_\mathcal{L}$ to support full reflexivity appear in Figure 4.

In core $\lambda_\mathcal{L}$ language universal types and map types cannot be the argument to **typecase**. The full language introduces distinguished labels to represent constructors of universal types and map types. The kinds of the distinguished labels are shown in Figure 5. These types now become syntactic sugar for applications of the appropriate labels, as shown in Figure 7. These new distinguished labels require new forms of abstractions in the type level; for labels ($\lambda\iota{:}\mathrm{L}(\kappa).\tau$), for label sets ($\lambda s{:}\mathrm{Ls}.\tau$) and for kinds ($\Lambda\chi.\tau$). This addition is also reflected at the kind level: Kinds include the kinds of the core $\lambda_\mathcal{L}$, kinds for label abstractions ($\mathrm{L}(\kappa_1) \rightarrow \kappa_2$), kinds for label set constructors ($\mathrm{Ls} \rightarrow \kappa$) and finally universal kinds ($\forall\chi.\kappa$), which are the kinds of kind abstractions in the type level.

There is one implication in 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. The type equivalence relation encodes the fact that they are equivalent to certain simpler types. The interesting equivalences are given in Figure 6. Notice that polykinded types do not have a label constructor in Figure 7. At run time, closed polykinded types will always be reduced to one of the other type forms.

Although **typecase** allows a programmer to determine if a label matches one of a given set of labels, it does not provide her with a way to learn about new labels. Therefore, full $\lambda_\mathcal{L}$ language introduces an operator **lindex**, which returns the integer associated with its argument label constant. This operator provides the programmer a way to

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

Figure 6: Polykinded type equivalences

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

Figure 7: Syntactic sugar for types

distinguish between labels at run time. The rule for **lindex** is straightforward.

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

Another addition is that of a label set analysis operator **setcase**. Because the language of label sets is fixed, **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 that the language is extended with strings and operations for concatenation and conversion to/from integers.

$$
\begin{array}{l}
\mathbf{fix\ settostring}{:}\forall\alpha{:}\mathrm{Ls}.\ \mathsf{string}.\Lambda\alpha{:}\mathrm{Ls}. \\
\mathbf{setcase}\ \alpha \\
\quad \begin{array}{lcl}
\{ \quad \varnothing & \Rightarrow & \text{“”}, \\
\quad\ \cup & \Rightarrow & \Lambda s_1{:}\mathrm{Ls}.\Lambda s_2{:}\mathrm{Ls}. \\
 & & (\mathbf{settostring}[s_1])\ +\!\!+ \\
 & & \text{“ ”}\ +\!\!+(\mathbf{settostring}[s_2]), \\
\quad\ \{\} & \Rightarrow & \Lambda\chi.\Lambda\iota{:}\mathrm{L}(\chi).\mathbf{int2string}(\mathbf{lindex}(\iota)), \\
\quad\ \mathcal{U} & \Rightarrow & \text{“U”} \\
\}
\end{array}
\end{array}
$$

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

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

In this rule, $e_{\{\}}$ must be able to take any label as its argument, whatever the kind of the label. Therefore this expression must be kind polymorphic.

5. HIGHER-ORDER ANALYSIS

Higher-order type analysis [31] is an extension of run-time analysis to types of higher-order kind. It is 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. For example, a generic "length" operation that determines the length of a list and the number of nodes in a tree must 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.

The result of higher-order analysis depends on the kind of the analyzed constructor. In previous systems [31, 12] a type-directed operation is defined as an interpretation of a type constructor. Type functions are mapped to term functions, type applications to term applications, and type variables to term variables. In that way, equivalences in the term language reflect equivalences in the type language. Even though the types $(\lambda\alpha{:}\star.\mathsf{int})\ \mathsf{bool}$ and bool are syntactically different, they are semantically the same type, and so analysis produces the same results.

However, in $\lambda_{\mathcal{L}}$, analysis is over the weak-head normal form of types. Because equal types have the same normal form, such equivalences are already preserved. Furthermore, because we know that all constructors of kind $\star \to \star$ are equivalent to type functions (by extensionality) we can encode the analysis of such a constructor as a polymorphic term function, whose body uses **typecase** to analyze a constructor of kind $\star$. Generalizing to all kinds, we can encode higher-order analysis with first-order analysis.

For example, suppose f is an open polytypic operation of type $\forall s{:}\mathrm{Ls}.\forall\beta{:}(\star\restriction s).(s \Rightarrow \tau'\restriction s\cup\mathcal{L}) \to \tau'\beta$. Say we want to use the instance of f for the type τ of kind $\star \to \star$, and that $\mathcal{L}$ contains all the labels in τ. To do so, we modify the call site of f to be a polymorphic function, because that is the interpretation of type functions. This function abstracts the type argument β and a branch x as the interpretation of β. It then creates a new label for β and passes a branch to f that maps the new label to the interpretation of β. That way, no matter what type β is instantiated with, its interpretation will always be x.

$$\begin{array}{l}\Lambda\beta{:}\star\restriction\mathcal{L}.\lambda x{:}\tau'\beta.\mathbf{new}\ \iota{:}\star = \beta\ \mathbf{in}\\ \quad \{\!\{f\ [\{\iota\}]\ [\tau\ \iota]\{\iota \Rightarrow \{\!\{x : \tau'\}\!\}^{+}_{\iota}\} : \tau'\}\!\}^{-}_{\iota}\end{array}$$

6. EXTENSIONS

Default branches. One difficulty of working with $\lambda_{\mathcal{L}}$ is that **typecase** must always have a branch for the label of its argument. We showed earlier how to work around this using higher-order coercions or first-class maps. However, in some cases it is more natural to provide *default branches* that apply when no other branches match a label. To do so we add another form of map $\{\_ \Rightarrow e\}$ with a domain of all labels. With this extension, type variables restricted by $\mathcal{U}$ are not parametric.

$$\frac{\Delta \vdash \tau' : \star \to \star \qquad \Delta;\Gamma \vdash e : \forall\chi.\forall\alpha{:}\chi\restriction\mathcal{U}.\tau'\langle\alpha : \chi\restriction\mathcal{U}\rangle \mid \Sigma}{\Delta;\Gamma \vdash \{\_ \Rightarrow e\} : \mathcal{U} \Rightarrow \tau'\restriction\mathcal{U} \mid \Sigma}$$

This branch matches labels of *any* kind, so its type depends on the kind of the matched label. Therefore the type is kind polymorphic. Because of this polymorphism, within $\lambda_{\mathcal{L}}$ there are no reasonable terms that could be a default branch. However, with addition linguistic mechanisms such as exceptions, these default branches provide another way to treat new type names.

Recursive uncoercions. New types in $\lambda_{\mathcal{L}}$ may be recursively defined. However, if they are, higher-order coercions cannot completely eliminate a new label from the type of an expression. Instead, the coercion will unroll the type once, leaving an occurrence of the new label. It is possible to use first-order coercions to recursively remove all occurrences of a new type, but this will result in unnecessarily decomposing and rebuilding the data structure. Because coercions have no computational content, it is reasonable to provide a primitive operator $[\![\cdot]\!]^{-}_{l=\tau}$ for this uncoercing.

$$\frac{\Delta;\Gamma \vdash e : \tau' l \mid \Sigma \qquad l{:}\kappa = \tau \in \Sigma \qquad \Delta \vdash \tau \mid \mathcal{L}\cup\{l\}}{\Delta;\Gamma \vdash [\![e : \tau']\!]^{-}_{l=\tau} : \exists\alpha{:}\kappa\restriction(\mathcal{L}\cup\{\ell_{\mathsf{int}}\}).\tau'\alpha \mid \Sigma}$$

Because it is impossible to know statically what the exact shape of e is, the unrolled type of e is hidden using an existential type. Where the type "bottoms out" we use int, although we could use any other type. For example, if $\iota = 1 + (\mathsf{int} \times \iota)$, then the following list could be uncoerced as follows:

$$\begin{array}{l}[\![\{\!\{\mathbf{inr}\,\langle 1, \{\!\{\mathbf{inr}\,\langle 3, \{\!\{\mathbf{inl}\,\langle\rangle\}\!\}^{+}_{\iota}\rangle\}\!\}^{+}_{\iota}\rangle\}\!\}^{+}_{\iota} : \lambda\alpha{:}\star.\alpha]\!]^{-}_{\iota=1+\mathsf{int}\times\iota} \mapsto^{*}\\ \quad [1 + \mathsf{int} \times (1 + \mathsf{int} \times (1 + \mathsf{int})), \mathbf{inr}\,\langle 1, \mathbf{inr}\,\langle 3, \mathbf{inl}\,\langle\rangle\rangle\rangle]\ \mathbf{as}\\ \qquad \exists\alpha{:}\star\restriction\{\ell_{\times}, \ell_{+}, \ell_{1}, \ell_{\mathsf{int}}\}.\alpha\end{array}$$

The resulting existential package could then be opened and its contents used as the arguments to a type-directed operation that cannot handle the label ι.

Record and variant types. Current systems for type-directed programming have trouble with record and variant types, because of the names of fields and constructors. Often these systems translate these types into some internal representation before analysis [2]. Because labels are an integral part of $\lambda_{\mathcal{L}}$, with a small extension we can use them to represent these types natively.

The extension that we need for record and variant types is *finite type maps* from from labels to types of kind $\star$. Finite type maps are new syntactic category with their own form of abstraction and application in both the type and term languages, as well as finite map analysis. Rules analogous to those for label set subsumption, membership and equality can be defined for these finite maps.

$$\begin{array}{lcl}\mathcal{M} & ::= & \varnothing \mid \{l : \tau\} \mid m \mid \mathcal{M}_1 \cup \mathcal{M}_2\\ \kappa & ::= & \ldots \mid \textsc{Map} \to \kappa\\ \tau & ::= & \ldots \mid \lambda m{:}\textsc{Map}.\tau \mid \tau\ \mathcal{M}\\ e & ::= & \ldots \mid \Lambda m{:}\textsc{Map}.e \mid e[\mathcal{M}] \mid \mathbf{mapcase}\ \mathcal{M}\ \phi\\ & \mid & \varnothing \mid \{l = e\} \mid e_1 \circ e_2 \mid e.l\\ \phi & ::= & \{\varnothing \Rightarrow e_{\varnothing},\ \{\} \Rightarrow e_{\{\}},\ \cup \Rightarrow e_{\cup}\}\end{array}$$

The distinguished label ℓ_{rec} of kind $(\textsc{Map} \to \star)$ forms record types from finite maps. As with many versions of records, these types are equivalent up to permutation. Record terms are formed from empty records $\varnothing$, singletons $\{l = e\}$, or concatenation $e_1 \circ e_2$. If l is in the domain of the record type, the record projection $e.l$ is well-formed. Because we provide abstractions over finite maps, these records get a form of row polymorphism [24] for free. It is straightforward to develop similar extensions for variants.

The key difference between records and the branches used by **typecase** is that for a record, each label must be of kind $\star$. If arbitrarily-kinded labels were allowed, then code analyzing record types would need to be kind polymorphic, limiting its usefulness.

7. 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 [11]. 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 [28]. Weirich [31] extended run-time analysis to higher-order type constructors following the work of Hinze [12].

Generic programming uses the structure of datatypes to generate specialized operations at compile time. The Charity language [3] automatically generates folds for datatypes. PolyP [15] is an extension of Haskell that allows the definition of polytypic operations based on positive, regular datatypes. Functorial ML [17] bases polytypic operations on the composition of functors, and has lead to the programming language FISh [16]. Generic Haskell [2], following the work of Hinze [12] 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 [30] implement overloading by defining classes of types that have instances for a set of polytypic operations. Hinze and Peyton Jones [13] explored an extension to automatically derive type class instances by looking at the underlying structure of new types. Dependency-style Generic Haskell [20] 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 [21] and Haskell [23] rely on datatype generativity in type inference. Modern module systems also provide generative types [5]. 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 [10] or Stone and Harper [27]) 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. Explicit coercions also make sense in conjunction with type-directed programming because even if the definition is known, the distinction should still be made during dynamic type analysis.

A few researchers have considered the combination of generative types with forms of dynamic type analysis. Glew's [8] source language dynamically checks predeclared subtyping relationships between type names. Lämmel and Peyton Jones [18] used dynamic type equality checks to implement a number of polytypic iterators. Rossberg's λ_N calculus [26] 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 pre-computed expression. However, his coercions have a different semantics from ours. Higher-order coercions are reminiscent of the colored brackets of Grossman et al. [9], which are also used by Leifer et al. [19] to preserve type generativity when marshalling.

8. DISCUSSION

In conclusion, the $\lambda_{\mathcal{L}}$ language provides a good way to understand the properties of both nominal and structural type analysis. 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.

In the design of $\lambda_{\mathcal{L}}$, we explored many alternatives to simplify the language. For example, we tried combining labels and label sets into the same syntactic category as types, thereby eliminating the need for separate abstraction forms. However, this combination dramatically increases the complexity of the semantics. The fact that this change allows **new** expressions to create new names for not just types, but label sets and even labels, complicates the process of determining the appropriate set of labels used in a type constructor.

Aside from developing a usable source language, there are a number 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 possessing more sophisticated type generativity. Furthermore, type analysis is especially useful for applications such as marshalling and dynamic loading, so it would useful to develop a distributed calculus based upon $\lambda_{\mathcal{L}}$. 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.

Finally, to increase the expressiveness of the core language, we plan to extend it in two ways. First, **typecase** makes restrictions on all labels that appear in its argument so that it can express catamorphisms 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 $\mathsf{int} \to \mathsf{int}$ or $\mathsf{int} \to \mathsf{int} \to \mathsf{int}$. 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 [14, 6] can statically enforce that tree-structured data has a very particular form.

Furthermore, it is also important to add type-level analysis of types to the language. As shown in past work, 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. [28] to include first-class maps from labels to types.

Acknowledgments

Thanks to Steve Zdancewic, Benjamin Pierce, and Andreas Rossberg for helpful discussion.

9. 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] 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.

[6] V. Gapayev 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.

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

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

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

[10] 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.

[11] 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.

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

[13] 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.

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

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

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

[24] 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`.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

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

APPENDIX

A. LANGUAGE

A.1 Syntax

$$
\begin{array}{lll}
\textit{Kinds} & \kappa ::= & \chi \mid \star \mid \kappa_1 \to \kappa_2 \\
 & \mid & \mathrm{Ls} \to \kappa \mid \mathrm{L}(\kappa_1) \to \kappa_2 \mid \forall\chi.\kappa \\
\textit{Labels} & l ::= & \ell_i^\kappa \mid \iota \\
\textit{Label sets} & \mathcal{L} ::= & \varnothing \mid \{l\} \mid s \mid \mathcal{L}_1 \cup \mathcal{L}_2 \mid \mathcal{U} \\
\textit{Types} & \tau ::= & \alpha \mid \lambda\alpha{:}\kappa.\tau \mid \tau_1\tau_2 \mid \lambda\iota{:}\mathrm{L}(\kappa).\tau \mid \tau\ \hat{l} \\
 & \mid & \lambda s{:}\mathrm{Ls}.\tau \mid \tau\ \mathcal{L} \mid \Lambda\chi.\tau \mid \tau[\kappa] \\
 & \mid & \tau'\langle\tau : \kappa \restriction \mathcal{L}\rangle \mid l \\
\textit{Terms} & e ::= & x \mid \lambda x{:}\tau.e \mid e_1e_2 \mid \mathbf{fix}\ x{:}\tau.e \mid i \\
 & \mid & \mathbf{new}\ \iota{:}\kappa = \tau\ \mathbf{in}\ e \mid \{\!\{e\}\!\}^{\pm}_{l=\tau} \mid \{\!\{e : \tau\}\!\}^{\pm}_{l=\tau_2} \\
 & \mid & \mathbf{typecase}\ \tau\ e \mid \mathbf{setcase}\ \mathcal{L}\ \theta \\
 & \mid & \mathbf{lindex}\ l \mid \Lambda\alpha{:}\kappa\restriction\mathcal{L}.e \mid e[\tau] \\
 & \mid & \Lambda\iota{:}\mathrm{L}(\kappa).e \mid e[l] \mid \Lambda s{:}\mathrm{Ls}.e \mid e[\mathcal{L}] \\
 & \mid & \Lambda\chi.e \mid e[\kappa] \mid \varnothing \mid \{l \Rightarrow e\} \mid e_1 \bowtie e_2 \\
\textit{Setcase branches} & \theta ::= & \{\varnothing \Rightarrow e_\varnothing,\ \{\} \Rightarrow e_{\{\}}, \cup \Rightarrow e_\cup,\ \mathcal{U} \Rightarrow e_\mathcal{U}\ \}
\end{array}
$$

A.2 Judgments

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	$\Delta \vdash \tau \mid \mathcal{L}$
Type equivalence	$\Delta \vdash \tau_1 = \tau_2 : \kappa$
Term well-formedness	$\Delta; \Gamma \vdash e : \tau \mid \Sigma$
Dynamic Judgments	
Small-step evaluation	$\mathcal{L}; e \mapsto \mathcal{L}'; e'$
Weak-head reduction	$\Delta \vdash \tau \Downarrow \tau'$
Weak-head normalization	$\Delta \vdash \tau \downarrow \tau'$
Label set reduction	$\mathcal{L}_1 \downarrow \mathcal{L}_2$
Path conversion	$\rho \rightsquigarrow p$

A.3 Static semantics

A.3.1 Kind well-formedness

$$\frac{\chi \in \Delta}{\Delta \vdash \chi}\ \textsf{wfk:var} \qquad \frac{}{\Delta \vdash \star}\ \textsf{wfk:type}$$

$$\frac{\Delta \vdash \kappa_1 \qquad \Delta \vdash \kappa_2}{\Delta \vdash \mathrm{L}(\kappa_1) \to \kappa_2}\ \textsf{wfk:larrow}$$

$$\frac{\Delta \vdash \kappa}{\Delta \vdash \mathrm{Ls} \to \kappa}\ \textsf{wfk:sarrow}$$

$$\frac{\Delta \vdash \kappa_1 \qquad \Delta \vdash \kappa_2}{\Delta \vdash \kappa_1 \to \kappa_2}\ \textsf{wfk:arrow} \qquad \frac{\Delta, \chi \vdash \kappa}{\Delta \vdash \forall\chi.\kappa}\ \textsf{wfk:all}$$

A.3.2 Label well-formedness

$$\frac{\vdash \Delta}{\Delta \vdash \ell_i^\kappa : \mathrm{L}(\kappa)}\ \textsf{wfl:const} \qquad \frac{\vdash \Delta \qquad \iota{:}\mathrm{L}(\kappa) \in \Delta}{\Delta \vdash \iota : \mathrm{L}(\kappa)}\ \textsf{wfl:var}$$

A.3.3 Label set well-formedness

$$\frac{}{\Delta \vdash \varnothing : \mathrm{Ls}}\ \textsf{wfls:empty} \qquad \frac{\Delta \vdash l : \mathrm{L}(\kappa)}{\Delta \vdash \{l\} : \mathrm{Ls}}\ \textsf{wfls:sing}$$

$$\frac{s{:}\mathrm{Ls} \in \Delta}{\Delta \vdash s : \mathrm{Ls}}\ \textsf{wfls:var}$$

$$\frac{\Delta \vdash \mathcal{L}_1 : \mathrm{Ls} \qquad \Delta \vdash \mathcal{L}_2 : \mathrm{Ls}}{\Delta \vdash \mathcal{L}_1 \cup \mathcal{L}_2 : \mathrm{Ls}}\ \textsf{wfls:union}$$

$$\frac{}{\Delta \vdash \mathcal{U} : \mathrm{Ls}}\ \textsf{wfls:univ}$$

A.3.4 Type well-formedness

$$\frac{\alpha{:}\kappa \in \Delta}{\Delta \vdash \alpha : \kappa}\ \textsf{twf:var} \qquad \frac{\alpha{:}\kappa\restriction\mathcal{L} \in \Delta}{\Delta \vdash \alpha : \kappa}\ \textsf{twf:var-res}$$

$$\frac{\Delta \vdash \tau_1 : \kappa_1 \to \kappa_2 \qquad \Delta \vdash \tau_2 : \kappa_1}{\Delta \vdash \tau_1\tau_2 : \kappa_2}\ \textsf{twf:app}$$

$$\frac{\Delta, \alpha{:}\kappa_1 \vdash \tau : \kappa_2 \qquad \Delta \vdash \kappa_1}{\Delta \vdash \lambda\alpha{:}\kappa_1.\tau : \kappa_1 \to \kappa_2}\ \textsf{twf:abs}$$

$$\frac{\Delta \vdash \tau : \kappa \qquad \Delta \vdash \tau' : \star \to \star \qquad \Delta \vdash \mathcal{L} : \mathrm{Ls}}{\Delta \vdash \tau'\langle\tau : \kappa\restriction\mathcal{L}\rangle : \star}\ \textsf{twf:polyk}$$

$$\frac{\Delta \vdash l : \mathrm{L}(\kappa)}{\Delta \vdash l : \kappa}\ \textsf{twf:ltype}$$

`$Id: itaname.tex 240 2004-04-01 21:25:19Z sweirich $`

$$\frac{\Delta, \chi \vdash \tau : \kappa}{\Delta \vdash \Lambda\chi.\tau : \forall\chi.\kappa} \text{ twf:kabs}$$

$$\frac{\Delta \vdash \tau : \forall\chi.\kappa_2 \qquad \Delta \vdash \kappa_1}{\Delta \vdash \tau[\kappa_1] : \kappa_2[\kappa_1/\chi]} \text{ twf:kapp}$$

$$\frac{\Delta, \iota{:}\mathrm{L}(\kappa_1) \vdash \tau : \kappa_2 \qquad \Delta \vdash \kappa_1 \qquad \iota \notin \mathcal{L}}{\Delta \vdash \lambda\iota{:}\mathrm{L}(\kappa_1).\tau : \mathrm{L}(\kappa_1) \to \kappa_2} \text{ twf:labs}$$

$$\frac{\Delta \vdash \tau : \mathrm{L}(\kappa_1) \to \kappa_2 \qquad \Delta \vdash l : \mathrm{L}(\kappa_1)}{\Delta \vdash \tau\hat{l} : \kappa_2} \text{ twf:lapp}$$

$$\frac{\Delta, s{:}\mathrm{Ls} \vdash \tau : \kappa \qquad s \notin \mathcal{L}}{\Delta \vdash \lambda s{:}\mathrm{Ls}.\tau : \mathrm{Ls} \to \kappa} \text{ twf:sabs}$$

$$\frac{\Delta \vdash \tau : \mathrm{Ls} \to \kappa \qquad \Delta \vdash \mathcal{L}_2 : \mathrm{Ls}}{\Delta \vdash \tau\mathcal{L}_2 : \kappa} \text{ twf:sapp}$$

A.3.5 Type label set analysis

$$\frac{\alpha{:}\kappa \in \Delta}{\Delta \vdash \alpha \,|\, \varnothing} \text{ tan:var} \qquad \frac{\alpha{:}\kappa \restriction \mathcal{L} \in \Delta}{\Delta \vdash \alpha \,|\, \mathcal{L}} \text{ tan:var-res}$$

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

$$\frac{\Delta, \alpha{:}\kappa_1 \vdash \tau \,|\, \kappa_2\mathcal{L}}{\Delta \vdash \lambda\alpha{:}\kappa_1.\tau \,|\, \mathcal{L}} \text{ tan:abs}$$

$$\frac{\Delta \vdash \tau \,|\, \mathcal{L}_1 \qquad \Delta \vdash \tau' \,|\, \mathcal{L}_2}{\Delta \vdash \tau'\langle \tau : \kappa \restriction \mathcal{L}\rangle \,|\, \mathcal{L}_1 \cup \mathcal{L}_2} \text{ tan:polyk}$$

$$\frac{}{\Delta \vdash l \,|\, \{l\}} \text{ tan:ltype}$$

$$\frac{\Delta, \chi \vdash \tau \,|\, \mathcal{L}}{\Delta \vdash \Lambda\chi.\tau \,|\, \mathcal{L}} \text{ tan:kabs} \qquad \frac{\Delta \vdash \tau \,|\, \mathcal{L}}{\Delta \vdash \tau[\kappa_1] \,|\, \mathcal{L}} \text{ tan:kapp}$$

$$\frac{\Delta, \iota{:}\mathrm{L}(\kappa_1) \vdash \tau \,|\, \mathcal{L}}{\Delta \vdash \lambda\iota{:}\mathrm{L}(\kappa_1).\tau \,|\, \mathcal{L}} \text{ tan:labs} \qquad \frac{\Delta \vdash \tau \,|\, \mathcal{L}}{\Delta \vdash \tau\hat{l} \,|\, \mathcal{L}} \text{ tan:lapp}$$

$$\frac{\Delta, s{:}\mathrm{Ls} \vdash \tau \,|\, \mathcal{L}}{\Delta \vdash \lambda s{:}\mathrm{Ls}.\tau \,|\, \mathcal{L}} \text{ tan:sabs} \qquad \frac{\Delta \vdash \tau \,|\, \mathcal{L}_1}{\Delta \vdash \tau\mathcal{L}_2 \,|\, \mathcal{L}_1} \text{ tan:sapp}$$

A.3.6 Label set subsumption

$$\frac{\Delta \vdash \mathcal{L} : \mathrm{Ls}}{\Delta \vdash \mathcal{L} \sqsubseteq \mathcal{L}} \text{ ss:refl}$$

$$\frac{\Delta \vdash \mathcal{L}_1 \sqsubseteq \mathcal{L}_2 \qquad \Delta \vdash \mathcal{L}_2 \sqsubseteq \mathcal{L}_3}{\Delta \vdash \mathcal{L}_1 \sqsubseteq \mathcal{L}_3} \text{ ss:trans}$$

$$\frac{\Delta \vdash \mathcal{L}_1 \sqsubseteq \mathcal{L} \qquad \Delta \vdash \mathcal{L}_2 \sqsubseteq \mathcal{L}}{\Delta \vdash \mathcal{L}_1 \cup \mathcal{L}_2 \sqsubseteq \mathcal{L}} \text{ ss:union-left}$$

$$\frac{\Delta \vdash \mathcal{L} \sqsubseteq \mathcal{L}_1 \qquad \Delta \vdash \mathcal{L}_2 : \mathrm{Ls}}{\Delta \vdash \mathcal{L} \sqsubseteq \mathcal{L}_1 \cup \mathcal{L}_2} \text{ ss:union-right1}$$

$$\frac{\Delta \vdash \mathcal{L} \sqsubseteq \mathcal{L}_2 \qquad \Delta \vdash \mathcal{L}_1 : \mathrm{Ls}}{\Delta \vdash \mathcal{L} \sqsubseteq \mathcal{L}_1 \cup \mathcal{L}_2} \text{ ss:union-right2}$$

$$\frac{\Delta \vdash \mathcal{L} : \mathrm{Ls}}{\Delta \vdash \varnothing \sqsubseteq \mathcal{L}} \text{ ss:empty} \qquad \frac{\Delta \vdash \mathcal{L} : \mathrm{Ls}}{\Delta \vdash \mathcal{L} \sqsubseteq \mathcal{U}} \text{ ss:univ}$$

A.3.7 Label set equivalence

$$\frac{\Delta \vdash \mathcal{L}_1 \sqsubseteq \mathcal{L}_2 \qquad \Delta \vdash \mathcal{L}_2 \sqsubseteq \mathcal{L}_1}{\Delta \vdash \mathcal{L}_1 = \mathcal{L}_2} \text{ seq:deriv}$$

A.3.8 Type equivalence

$$\frac{\Delta \vdash \tau : \kappa}{\Delta \vdash \tau = \tau : \kappa} \text{ teq:refl} \qquad \frac{\Delta \vdash \tau_1 = \tau_2 : \kappa}{\Delta \vdash \tau_2 = \tau_1 : \kappa} \text{ teq:sym}$$

$$\frac{\Delta \vdash \tau_1 = \tau_2 : \kappa \qquad \Delta \vdash \tau_2 = \tau_3 : \kappa}{\Delta \vdash \tau_1 = \tau_3 : \kappa} \text{ teq:trans}$$

$$\frac{\Delta \vdash \lambda\alpha{:}\kappa_1.\tau_1 : \kappa_1 \to \kappa_2 \qquad \Delta \vdash \tau_2 : \kappa_1}{\Delta \vdash (\lambda\alpha{:}\kappa_1.\tau_1)\tau_2 = \tau_1[\tau_2/\alpha] : \kappa_2} \text{ teq:abs-beta}$$

$$\frac{\Delta \vdash \tau : \kappa_1 \to \kappa_2}{\Delta \vdash \lambda\alpha{:}\kappa_1.\tau\alpha = \tau : \kappa_1 \to \kappa_2} \text{ teq:abs-eta}$$

$$\frac{\Delta \vdash \tau_1 = \tau_3 : \kappa_1 \to \kappa_2 \qquad \Delta \vdash \tau_2 = \tau_4 : \kappa_1}{\Delta \vdash \tau_1\tau_2 = \tau_3\tau_4 : \kappa_2} \text{ teq:app-con}$$

$$\frac{\Delta, \alpha{:}\kappa_1 \vdash \tau_1 = \tau_2 : \kappa_2 \qquad \Delta \vdash \kappa_1}{\Delta \vdash \lambda\alpha{:}\kappa_1.\tau_1 = \lambda\alpha{:}\kappa_1.\tau_2 : \kappa_1 \to \kappa_2} \text{ teq:abs-con}$$

$$\frac{\Delta \vdash \tau_1 = \tau_2 : \forall\chi.\kappa_2 \qquad \Delta \vdash \kappa_1}{\Delta \vdash \tau_1[\kappa_1] = \tau_2[\kappa_1] : \kappa_2} \text{ teq:kapp-con}$$

$$\frac{\Delta, \chi \vdash \tau_1 = \tau_2 : \kappa_2}{\Delta \vdash \Lambda\chi.\tau_1 = \Lambda\chi.\tau_2 : \kappa_1 \to \kappa_2} \text{ teq:kabs-con}$$

$$\frac{\Delta \vdash \Lambda\chi.\tau_1 : \forall\chi.\kappa_2 \qquad \Delta \vdash \kappa_1}{\Delta \vdash (\Lambda\chi.\tau_1)[\kappa_1] = \tau_2[\kappa_1/\chi] : \kappa_2[\kappa_1/\chi]} \text{ teq:kabs-beta}$$

$$\frac{\Delta \vdash \tau : \forall\chi.\kappa}{\Delta \vdash \Lambda\chi.\tau[\chi] = \tau : \forall\chi.\kappa} \text{ teq:kabs-eta}$$

$$\frac{\Delta \vdash \tau_1 = \tau_2 : \mathrm{L}(\kappa_1) \to \kappa_2 \qquad \Delta \vdash l : \mathrm{L}(\kappa_1)}{\Delta \vdash \tau_1\hat{l} = \tau_2\hat{l} : \kappa_2} \text{ teq:lapp-con}$$

$$\frac{\Delta, \iota{:}\mathrm{L}(\kappa_1) \vdash \tau_1 = \tau_2 : \kappa_2}{\Delta \vdash \lambda\iota{:}\mathrm{L}(\kappa_1).\tau_1 = \lambda\iota{:}\mathrm{L}(\kappa_1).\tau_2 : \mathrm{L}(\kappa_1) \to \kappa_2} \text{ teq:labs-con}$$

$$\frac{\Delta \vdash \lambda\iota{:}\mathrm{L}(\kappa_1).\tau : \mathrm{L}(\kappa_1) \to \kappa_2 \qquad \Delta \vdash l : \mathrm{L}(\kappa_1)}{\Delta \vdash (\lambda\iota{:}\mathrm{L}(\kappa_1).\tau)\hat{l} = \tau[l/\iota] : \kappa_2} \text{ teq:labs-beta}$$

$$\frac{\Delta \vdash \tau : \mathrm{L}(\kappa_1) \to \kappa_2}{\Delta \vdash \lambda\iota{:}\mathrm{L}(\kappa_1).\tau\iota = \tau : \mathrm{L}(\kappa_1) \to \kappa_2} \text{ teq:labs-eta}$$

$$\frac{\Delta \vdash \tau_1 = \tau_2 : \mathrm{Ls} \to \kappa \qquad \Delta \vdash \mathcal{L}_1 = \mathcal{L}_2}{\Delta \vdash \tau_1\mathcal{L}_1 = \tau_2\mathcal{L}_2 : \kappa}\ \textsf{teq:sapp-con}$$

$$\frac{\Delta, s{:}\mathrm{Ls} \vdash \tau_1 = \tau_2 : \kappa}{\Delta \vdash \lambda s{:}\mathrm{Ls}.\tau_1 = \lambda s{:}\mathrm{Ls}.\tau_2 : \mathrm{Ls} \to \kappa}\ \textsf{teq:sabs-con}$$

$$\frac{\Delta \vdash \lambda s{:}\mathrm{Ls}.\tau : \mathrm{Ls} \to \kappa \qquad \Delta \vdash \mathcal{L} : \mathrm{Ls}}{\Delta \vdash (\lambda s{:}\mathrm{Ls}.\tau)\mathcal{L} = \tau[\mathcal{L}/s] : \kappa}\ \textsf{teq:sabs-beta}$$

$$\frac{\Delta \vdash \tau : \mathrm{Ls} \to \kappa}{\Delta \vdash \lambda s{:}\mathrm{Ls}.\tau s = \tau : \mathrm{Ls} \to \kappa}\ \textsf{teq:sabs-eta}$$

$$\frac{\begin{array}{c}\Delta \vdash \tau_1 = \tau_2 : \kappa \\ \Delta \vdash \tau_1' = \tau_2' : \star \to \star \qquad \Delta \vdash \mathcal{L}_1 = \mathcal{L}_2\end{array}}{\Delta \vdash \tau_1'\langle \tau_1 : \kappa \upharpoonright \mathcal{L}_1\rangle = \tau_2'\langle \tau_2 : \kappa \upharpoonright \mathcal{L}_2\rangle : \star}\ \textsf{teq:polyk-con}$$

$$\frac{\Delta \vdash \tau : \star \qquad \Delta \vdash \tau' : \star \to \star \qquad \Delta \vdash \mathcal{L} : \mathrm{Ls}}{\Delta \vdash \tau'\langle \tau : \star \upharpoonright \mathcal{L}\rangle = \tau'\tau : \star}\ \textsf{teq:polyk-type}$$

$$\frac{\begin{array}{c}\Delta \vdash \tau : \kappa_1 \to \kappa_2 \\ \Delta \vdash \tau' : \star \to \star \qquad \Delta \vdash \mathcal{L} : \mathrm{Ls}\end{array}}{\begin{array}{c}\Delta \vdash \tau'\langle \tau : \kappa_1 \to \kappa_2) \upharpoonright \mathcal{L}\rangle = \\ \forall \alpha{:}\kappa \upharpoonright \mathcal{L}.\tau'\langle \tau\alpha : \kappa_2 \upharpoonright \mathcal{L}\rangle : \star\end{array}}\ \textsf{teq:polyk-a}$$

$$\frac{\begin{array}{c}\Delta \vdash \tau : \mathrm{L}(\kappa_1) \to \kappa_2 \\ \Delta \vdash \tau' : \star \to \star \qquad \Delta \vdash \mathcal{L} : \mathrm{Ls}\end{array}}{\begin{array}{c}\Delta \vdash \tau'\langle \tau : \mathrm{L}(\kappa_1) \to \kappa_2 \upharpoonright \mathcal{L}\rangle = \\ \forall \iota{:}\mathrm{L}(\kappa_1).\tau'\langle \tau\iota : \kappa_2 \upharpoonright \mathcal{L}\rangle : \star\end{array}}\ \textsf{teq:polyk-la}$$

$$\frac{\begin{array}{c}\Delta \vdash \tau : \mathrm{Ls} \to \kappa \\ \Delta \vdash \tau' : \star \to \star \qquad \Delta \vdash \mathcal{L} : \mathrm{Ls}\end{array}}{\Delta \vdash \tau'\langle \tau : \mathrm{Ls} \to \kappa \upharpoonright \mathcal{L}\rangle = \forall s{:}\mathrm{Ls}.\tau'\langle \tau s : \kappa \upharpoonright \mathcal{L}\rangle : \star}\ \textsf{teq:polyk-sa}$$

$$\frac{\Delta \vdash \tau : \forall \chi.\kappa \qquad \Delta \vdash \tau' : \star \to \star \qquad \Delta \vdash \mathcal{L} : \mathrm{Ls}}{\Delta \vdash \tau'\langle \tau : \forall \chi.\kappa \upharpoonright \mathcal{L}\rangle = \forall \chi.\tau'\langle \tau[\chi] : \kappa \upharpoonright \mathcal{L}\rangle : \star}\ \textsf{teq:polyk-all}$$

A.3.9 Term well-formedness

$$\frac{x{:}\tau \in \Gamma}{\Delta; \Gamma \vdash x : \tau \mid \Sigma}\ \textsf{wf:var} \qquad \frac{}{\Delta; \Gamma \vdash i : \ell_{\mathsf{int}} \mid \Sigma}\ \textsf{wf:int}$$

$$\frac{\Delta \vdash \tau_1 : \star \qquad \Delta; \Gamma, x{:}\tau_1 \vdash e : \tau_2 \mid \Sigma}{\Delta; \Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2 \mid \Sigma}\ \textsf{wf:abs}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \mid \Sigma \qquad \Delta; \Gamma \vdash e_2 : \tau_1 \mid \Sigma}{\Delta; \Gamma \vdash e_1 e_2 : \tau_2 \mid \Sigma}\ \textsf{wf:app}$$

$$\frac{\Delta \vdash \tau : \star \qquad \Delta; \Gamma, x{:}\tau \vdash e : \tau \mid \Sigma}{\Delta; \Gamma \vdash \mathbf{fix}\ x{:}\tau.e : \tau \mid \Sigma}\ \textsf{wf:fix}$$

$$\frac{\begin{array}{c}\Delta, \iota{:}\mathrm{L}(\kappa); \Gamma \vdash e : \tau_2 \mid \Sigma, \iota{:}\kappa = \tau_1 \\ \Delta, \iota{:}\mathrm{L}(\kappa) \vdash \tau_1 : \kappa \qquad \iota \notin \tau_2\end{array}}{\Delta; \Gamma \vdash \mathbf{new}\ \iota{:}\kappa = \tau_1\ \mathbf{in}\ e : \tau_2 \mid \Sigma}\ \textsf{wf:new}$$

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

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

$$\frac{\Delta; \Gamma \vdash e : \tau'\ \tau \mid \Sigma \qquad l{:}\kappa = \tau \in \Sigma}{\Delta; \Gamma \vdash \{\!\{e : \tau'\}\!\}^{+}_{l=\tau} : \tau'\ l \mid \Sigma}\ \textsf{wf:hin}$$

$$\frac{\Delta; \Gamma \vdash e : \tau'\ l \mid \Sigma \qquad l{:}\kappa = \tau \in \Sigma}{\Delta; \Gamma \vdash \{\!\{e : \tau'\}\!\}^{-}_{l=\tau} : \tau'\ \tau \mid \Sigma}\ \textsf{wf:hout}$$

$$\frac{\Delta; \Gamma \vdash e : \tau_1 \mid \Sigma \qquad \Delta \vdash \tau_1 = \tau_2 : \star}{\Delta; \Gamma \vdash e : \tau_2 \mid \Sigma}\ \textsf{wf:weak}$$

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

$$\frac{\begin{array}{c}\Delta \vdash \mathcal{L} : \mathrm{Ls} \\ \Delta \vdash l : \mathrm{L}(\kappa) \qquad \Delta; \Gamma \vdash e : \tau'\langle l : \kappa \upharpoonright \mathcal{L}\rangle \mid \Sigma\end{array}}{\Delta; \Gamma \vdash \{l \Rightarrow e\} : \{l\} \Rightarrow \tau' \upharpoonright \mathcal{L} \mid \Sigma}\ \textsf{wf:env-branch}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash e_1 : \mathcal{L}_1 \Rightarrow \tau' \upharpoonright \mathcal{L} \mid \Sigma \\ \Delta; \Gamma \vdash e_2 : \mathcal{L}_2 \Rightarrow \tau' \upharpoonright \mathcal{L} \mid \Sigma\end{array}}{\Delta; \Gamma \vdash e_1 \bowtie e_2 : \mathcal{L}_1 \cup \mathcal{L}_2 \Rightarrow \tau' \upharpoonright \mathcal{L} \mid \Sigma}\ \textsf{wf:env-join}$$

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

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

$$\frac{\Delta \vdash l : \mathrm{L}(\kappa)}{\Delta; \Gamma \vdash \mathbf{lindex}\ l : \ell_{\mathsf{int}} \mid \Sigma}\ \textsf{wf:lindex}$$

$$\frac{\Delta \vdash \mathcal{L} : \mathrm{Ls} \qquad \Delta, \alpha{:}\kappa \upharpoonright \mathcal{L}; \Gamma \vdash e : \tau \mid \Sigma}{\Delta; \Gamma \vdash \Lambda \alpha{:}\kappa \upharpoonright \mathcal{L}.e : \forall \alpha{:}\kappa \upharpoonright \mathcal{L}.\tau \mid \Sigma}\ \textsf{wf:tabs}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash e : \forall \alpha{:}\kappa \upharpoonright \mathcal{L}.\tau_1 \mid \Sigma \\ \Delta \vdash \tau_2 : \kappa \qquad \Delta \vdash \tau_2 \mid \mathcal{L}' \qquad \Delta \vdash \mathcal{L}' \sqsubseteq \mathcal{L}\end{array}}{\Delta; \Gamma \vdash e[\tau_2] : \tau_1[\tau_2/\alpha] \mid \Sigma}\ \textsf{wf:tapp}$$

$$\frac{\Delta, \iota{:}\mathrm{L}(\kappa); \Gamma \vdash e : \tau \mid \Sigma \qquad \iota \notin \Sigma}{\Delta; \Gamma \vdash \Lambda \iota{:}\mathrm{L}(\kappa).e : \forall \iota{:}\mathrm{L}(\kappa).\tau \mid \Sigma}\ \textsf{wf:labs}$$

$$\frac{\Delta; \Gamma \vdash e : \forall \iota{:}\mathrm{L}(\kappa).\tau \mid \Sigma \qquad \Delta \vdash l : \mathrm{L}(\kappa)}{\Delta; \Gamma \vdash e[\hat{l}] : \tau[l/\iota] \mid \Sigma}\ \textsf{wf:lapp}$$

$$\frac{\Delta, s{:}\mathrm{Ls}; \Gamma \vdash e : \tau \mid \Sigma \qquad s \notin \Sigma}{\Delta; \Gamma \vdash \Lambda s{:}\mathrm{Ls}.e : \forall s.\tau \mid \Sigma}\ \textsf{wf:sabs}$$

$$\frac{\Delta; \Gamma \vdash e : \forall s{:}\mathrm{Ls}.\tau \mid \Sigma \qquad \Delta \vdash \mathcal{L} : \mathrm{Ls}}{\Delta; \Gamma \vdash e[\mathcal{L}] : \tau[\mathcal{L}/s] \mid \Sigma}\ \textsf{wf:sapp}$$

$$\frac{\Delta, \chi; \Gamma \vdash e : \tau \mid \Sigma}{\Delta; \Gamma \vdash \Lambda \chi.e : \forall \chi.\tau \mid \Sigma}\ \textsf{wf:kabs}$$

$$\frac{\Delta; \Gamma \vdash e : \forall \chi.\tau \mid \Sigma \qquad \Delta \vdash \kappa}{\Delta; \Gamma \vdash e[\kappa] : \tau[\kappa/\alpha] \mid \Sigma}\ \textsf{wf:kapp}$$

A.4 Dynamic semantics

$$\begin{array}{lll}
\textit{Values} & v ::= & \lambda x{:}\sigma.e \mid \{\!\{v\}\!\}^{+}_{l=\tau} \mid i \\
 & \mid & \varnothing \mid \{l \Rightarrow e\} \mid v_1 \bowtie v_2 \\
 & \mid & \Lambda \alpha{:}\kappa \upharpoonright \mathcal{L}.e \mid \Lambda \iota{:}\mathrm{L}(\kappa).e \\
 & \mid & \Lambda s{:}\mathrm{Ls}.e \mid \Lambda \chi.e \\
\textit{Tycon paths} & \rho ::= & \bullet \mid \rho\ \tau \mid \rho\ l \mid \rho\ \mathcal{L} \mid \rho\ [\kappa] \\
\textit{Term paths} & p ::= & \bullet \mid p\ [\tau] \mid p\ [\hat{l}] \mid p[\mathcal{L}] \mid p\ [\kappa]
\end{array}$$

A.4.1 Weak-head reduction for types

$$\frac{}{\Delta \vdash (\lambda \alpha{:}\kappa.\tau_1)\tau_2 \Downarrow \tau_1[\tau_2/\alpha]}\ \textsf{whr:abs-beta}$$

$$\frac{\Delta, \alpha{:}\kappa \vdash \tau \Downarrow \tau'}{\Delta \vdash \lambda \alpha{:}\kappa.\tau \Downarrow \lambda \alpha{:}\kappa.\tau'}\ \textsf{whr:abs-con}$$

$$\frac{\Delta \vdash \tau_1 \Downarrow \tau_1'}{\Delta \vdash \tau_1 \tau_2 \Downarrow \tau_1' \tau_2}\ \textsf{whr:app-con}$$

$$\frac{}{\Delta \vdash (\lambda \iota{:}\mathrm{L}(\kappa).\tau)\hat{l} \Downarrow \tau[l/\iota]}\ \textsf{whr:labs-beta}$$

$$\frac{\Delta \vdash \tau \Downarrow \tau'}{\Delta \vdash \tau \hat{l} \Downarrow \tau' \hat{l}}\ \textsf{whr:lapp-con}$$

$$\frac{}{\Delta \vdash (\lambda s{:}\mathrm{Ls}.\tau)\mathcal{L} \Downarrow \tau[\mathcal{L}/s]}\ \textsf{whr:sabs-beta}$$

$$\frac{\Delta \vdash \tau \Downarrow \tau'}{\Delta \vdash \tau \mathcal{L} \Downarrow \tau' \mathcal{L}}\ \textsf{whr:sapp-con}$$

$$\frac{}{\Delta \vdash (\Lambda \chi.\tau)[\kappa] \Downarrow \tau[\kappa/\chi]}\ \textsf{whr:kabs-beta}$$

$$\frac{\Delta \vdash \tau \Downarrow \tau'}{\Delta \vdash \tau[\kappa] \Downarrow \tau'[\kappa]}\ \textsf{whr:kapp-con}$$

$$\frac{}{\Delta \vdash \tau' \langle \tau : \star \upharpoonright \mathcal{L} \rangle \Downarrow \tau' \tau}\ \textsf{whr:polyk-type}$$

$$\frac{}{\Delta \vdash \tau' \langle \tau : \kappa_1 \to \kappa_2 \upharpoonright \mathcal{L} \rangle \Downarrow \forall \alpha{:}\kappa_1 \upharpoonright \mathcal{L}.\tau' \langle \tau \alpha : \kappa_2 \upharpoonright \mathcal{L} \rangle}\ \textsf{whr:polyk-a}$$

$$\frac{}{\Delta \vdash \tau' \langle \tau : \mathrm{L}(\kappa_1) \to \kappa_2 \upharpoonright \mathcal{L} \rangle \Downarrow \forall \iota{:}\mathrm{L}(\kappa_1).\tau' \langle \tau \iota : \kappa_2 \upharpoonright \mathcal{L} \rangle}\ \textsf{whr:polyk-la}$$

$$\frac{}{\Delta \vdash \tau' \langle \tau : \mathrm{Ls} \to \kappa \upharpoonright \mathcal{L} \rangle \Downarrow \forall s{:}\mathrm{Ls}.\tau' \langle \tau s : \kappa \upharpoonright \mathcal{L} \rangle}\ \textsf{whr:polyk-sa}$$

$$\frac{}{\Delta \vdash \tau' \langle \tau : \forall \chi.\kappa \upharpoonright \mathcal{L} \rangle \Downarrow \forall \chi.\tau' \langle \tau[\chi] : \kappa \upharpoonright \mathcal{L} \rangle}\ \textsf{whr:polyk-all}$$

A.4.2 Weak-head normalization for types

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

$$\frac{\Delta \vdash \tau : \kappa_1 \to \kappa_2 \qquad \Delta, \alpha{:}\kappa_1 \vdash \tau\ \alpha \downarrow \tau'}{\Delta \vdash \tau \downarrow \lambda \alpha{:}\kappa_1.\tau}\ \textsf{whn:simple-con}$$

A.4.3 Reduction for label sets

$$\frac{\mathcal{L}_1 \downarrow \mathcal{L}_1'}{\mathcal{L}_1 \cup \mathcal{L}_2 \downarrow \mathcal{L}_1' \cup \mathcal{L}_2}\ \textsf{lsr:union-con1}$$

$$\frac{\mathcal{L}_2 \downarrow \mathcal{L}_2'}{\mathcal{L}_1 \cup \mathcal{L}_2 \downarrow \mathcal{L}_1 \cup \mathcal{L}_2'}\ \textsf{lsr:union-con2}$$

$$\frac{}{\varnothing \cup \mathcal{L} \downarrow \mathcal{L}}\ \textsf{lsr:union-empty} \qquad \frac{}{\mathcal{U} \cup \mathcal{L} \downarrow \mathcal{U}}\ \textsf{lsr:union-univ}$$

$$\frac{\forall \{\ell_j^{\kappa}\} \sqsubseteq \mathcal{L} \qquad i < j}{\mathcal{L} \cup \{\ell_i^{\kappa'}\} \downarrow \{\ell_i^{\kappa'}\} \cup \mathcal{L}}\ \textsf{lsr:union-swap}$$

$$\frac{}{(\mathcal{L}_1 \cup \mathcal{L}_2) \cup \mathcal{L}_3 \downarrow \mathcal{L}_1 \cup (\mathcal{L}_2 \cup \mathcal{L}_3)}\ \textsf{lsr:union-assoc}$$

$$\frac{}{\mathcal{L}_1 \cup \mathcal{L}_2 \downarrow \mathcal{L}_2 \cup \mathcal{L}_1}\ \textsf{lsr:union-comm}$$

A.4.4 Label set normal forms

$$\frac{}{\varnothing\ norm}\ \textsf{ln:empty} \qquad \frac{}{\mathcal{U}\ norm}\ \textsf{ln:univ}$$

$$\frac{}{\{\ell_i^{\kappa}\}\ norm}\ \textsf{ln:sing}$$

$$\frac{\forall \ell_j^{\kappa'} \in \mathcal{L}, i < j \qquad \mathcal{L}\ norm \qquad \mathcal{L} \neq \varnothing, \mathcal{U}}{\{\ell_i^{\kappa}\} \cup \mathcal{L}\ norm}\ \textsf{ln:union}$$

A.4.5 Path conversion

$$\frac{}{\bullet \rightsquigarrow \bullet}\ \textsf{pc:hole} \qquad \frac{\rho \rightsquigarrow p}{\rho\ \tau \rightsquigarrow p\ [\tau]}\ \textsf{pc:app}$$

$$\frac{\rho \rightsquigarrow p}{\rho\ \hat{l} \rightsquigarrow p\ [\hat{l}]}\ \textsf{pc:lapp} \qquad \frac{\rho \rightsquigarrow p}{\rho\ \mathcal{L} \rightsquigarrow p\ [\mathcal{L}]}\ \textsf{pc:sapp}$$

$$\frac{\rho \rightsquigarrow p}{\rho\ [\kappa] \rightsquigarrow p\ [\kappa]}\ \textsf{pc:inst}$$

A.4.6 Computation rules

$$\frac{}{\mathcal{L}; (\lambda x{:}\tau.e_1)v_2 \mapsto \mathcal{L}; e_1[v_2/x]}\ \textsf{ev:abs-beta}$$

$$\frac{}{\mathcal{L}; \mathbf{fix}\ x{:}\tau.e \mapsto \mathcal{L}; e[\mathbf{fix}\ x{:}\tau.e/x]}\ \textsf{ev:fix-beta}$$

$$\frac{}{\mathcal{L}; (\Lambda\alpha{:}\kappa \restriction \Sigma.e)[\tau] \mapsto \mathcal{L}; e[\tau/\alpha]}\ \textsf{ev:tabs-beta}$$

$$\frac{}{\mathcal{L}; (\Lambda\iota{:}\mathrm{L}(\kappa).e)[\hat{l}] \mapsto \mathcal{L}; e[l/\iota]}\ \textsf{ev:labs-beta}$$

$$\frac{}{\mathcal{L}; (\Lambda s{:}\mathrm{Ls}.e)[\mathcal{L}] \mapsto \mathcal{L}; e[\mathcal{L}/s]}\ \textsf{ev:sabs-beta}$$

$$\frac{}{\mathcal{L}; (\Lambda\chi.e)[\kappa] \mapsto \mathcal{L}; e[\kappa/\chi]}\ \textsf{ev:kabs-beta}$$

$$\frac{}{\mathcal{L}; \{\!\{ \{\!\{ v \}\!\}^{+}_{l=\tau} \}\!\}^{-}_{l=\tau} \mapsto \mathcal{L}; v}\ \textsf{ev:in-out}$$

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

$$\frac{\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\rho[\alpha]}{\mathcal{L}; \{\!\{ v : \tau' \}\!\}^{+}_{l=\tau} \mapsto \mathcal{L}; \{\!\{ \{\!\{ v : \lambda\alpha{:}\kappa.\rho[\tau] \}\!\}^{+}_{l=\tau} \}\!\}^{+}_{l=\tau}}\ \textsf{ev:hc-base}$$

$$\frac{\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\rho[\alpha]}{\mathcal{L}; \{\!\{ v : \tau' \}\!\}^{-}_{l=\tau} \mapsto \mathcal{L}; \{\!\{ \{\!\{ v : \lambda\alpha{:}\kappa.\rho[l] \}\!\}^{-}_{l=\tau} \}\!\}^{-}_{l=\tau}}\ \textsf{ev:hc-base-out}$$

$$\frac{\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\ell_{\mathsf{int}}}{\mathcal{L}; \{\!\{ i : \tau' \}\!\}^{\pm}_{l=\tau} \mapsto \mathcal{L}; i}\ \textsf{ev:hc-int}$$

$$\frac{\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\tau_1 \to \tau_2 \qquad \vdash \tau_1' = \tau_1[\tau/\alpha] : \star}{\begin{array}{l}\mathcal{L}; \{\!\{ \lambda x{:}\tau_1'.e : \tau' \}\!\}^{+}_{l=\tau} \mapsto \\ \mathcal{L}; \lambda x{:}(\tau_1[l/\alpha]).\{\!\{ e[\{\!\{ x : \lambda\alpha{:}\kappa.\tau_1 \}\!\}^{-}_{l=\tau}/x] : \lambda\alpha{:}\kappa.\tau_2 \}\!\}^{+}_{l=\tau}\end{array}}\ \textsf{ev:hc-a1}$$

$$\frac{\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\tau_1 \to \tau_2 \qquad \vdash \tau_1' = \tau_1[l/\alpha] : \star}{\begin{array}{l}\mathcal{L}; \{\!\{ \lambda x{:}\tau_1'.e : \tau' \}\!\}^{-}_{l=\tau} \mapsto \\ \mathcal{L}; \lambda x{:}(\tau_1[\tau/\alpha]).\{\!\{ e[\{\!\{ x : \lambda\alpha{:}\kappa.\tau_1 \}\!\}^{+}_{l=\tau}/x] : \lambda\alpha{:}\kappa.\tau_2 \}\!\}^{-}_{l=\tau}\end{array}}\ \textsf{ev:hc-a2}$$

$$\frac{\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L}_2}{\mathcal{L}; \{\!\{ \varnothing : \tau' \}\!\}^{\pm}_{l=\tau} \mapsto \mathcal{L}; \varnothing}\ \textsf{ev:hc-empty}$$

$$\frac{\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L}_2}{\begin{array}{l}\mathcal{L}; \{\!\{ \{\ell_i^\kappa \Rightarrow e'\} : \tau' \}\!\}^{\pm}_{l=\tau} \mapsto \\ \mathcal{L}; \{\ell_i^k \Rightarrow \{\!\{ e' : \lambda\alpha{:}\star.\tau'\langle \ell_i^k : \kappa \restriction \mathcal{L}_2 \rangle \}\!\}^{\pm}_{l=\tau}\}\end{array}}\ \textsf{ev:hc-sing}$$

$$\frac{\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\mathcal{L}_1 \cup \mathcal{L}_2 \Rightarrow \tau' \restriction \mathcal{L}}{\begin{array}{l}\mathcal{L}; \{\!\{ v_1 \bowtie v_2 : \tau' \}\!\}^{\pm}_{l=\tau} \mapsto \\ \mathcal{L}; \{\!\{ v_1 : \lambda\alpha{:}\kappa.\mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L} \}\!\}^{\pm}_{l=\tau} \bowtie \{\!\{ v_2 : \lambda\alpha{:}\kappa.\mathcal{L}_2 \Rightarrow \tau' \restriction \mathcal{L} \}\!\}^{\pm}_{l=\tau}\end{array}}\ \textsf{ev:hc-join}$$

$$\frac{\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\forall\beta{:}\kappa' \restriction \mathcal{L}'.\tau'}{\begin{array}{l}\mathcal{L}; \{\!\{ \Lambda\beta{:}\kappa' \restriction \mathcal{L}.e : \tau' \}\!\}^{\pm}_{l=\tau} \mapsto \\ \mathcal{L}; \Lambda\beta{:}\kappa' \restriction \mathcal{L}.\{\!\{ e : \lambda\alpha{:}\kappa.\tau' \}\!\}^{\pm}_{l=\tau}\end{array}}\ \textsf{ev:hc-tabs}$$

$$\frac{\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\forall\chi.\tau'}{\begin{array}{l}\mathcal{L}; \{\!\{ \Lambda\chi.e : \tau' \}\!\}^{\pm}_{l=\tau} \mapsto \\ \quad \mathcal{L}; \Lambda\chi.\{\!\{ e : \lambda\alpha{:}\kappa.\tau' \}\!\}^{\pm}_{l=\tau}\end{array}}\ \textsf{ev:hc-kabs}$$

$$\frac{\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\forall s{:}\mathrm{Ls}.\tau'}{\begin{array}{l}\mathcal{L}; \{\!\{ \Lambda s{:}\mathrm{Ls}.e : \tau' \}\!\}^{\pm}_{l=\tau} \mapsto \\ \quad \mathcal{L}; \Lambda s{:}\mathrm{Ls}.\{\!\{ e : \lambda\alpha{:}\kappa.\tau' \}\!\}^{\pm}_{l=\tau}\end{array}}\ \textsf{ev:hc-sabs}$$

$$\frac{\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\forall\iota{:}\mathrm{L}(\kappa').\tau'}{\begin{array}{l}\mathcal{L}; \{\!\{ \Lambda\iota{:}\mathrm{L}(\kappa').e : \tau' \}\!\}^{\pm}_{l=\tau} \mapsto \\ \quad \mathcal{L}; \Lambda\iota{:}\mathrm{L}(\kappa').\{\!\{ e : \lambda\alpha{:}\kappa.\tau' \}\!\}^{\pm}_{l=\tau}\end{array}}\ \textsf{ev:hc-labs}$$

$$\frac{\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\rho[l_1]}{\begin{array}{l}\mathcal{L}; \{\!\{ \{\!\{ v \}\!\}^{+}_{l_1=\tau_1} : \tau' \}\!\}^{\pm}_{l_2=\tau_2} \mapsto \\ \mathcal{L}; \{\!\{ \{\!\{ v : \lambda\alpha{:}\kappa.\rho[\tau_1] \}\!\}^{\pm}_{l_2=\tau_2} \}\!\}^{+}_{l_1=\tau_1}\end{array}}\ \textsf{ev:hc-color}$$

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

$$\frac{\vdash \tau \downarrow \rho[\ell_i^\kappa] \qquad \ell_i \notin v \qquad \{\_ \Rightarrow e'\} \in v}{\mathcal{L}; \mathbf{typecase}\ \tau\ v \mapsto \mathcal{L}; e'[\kappa][\ell_i^\kappa]}\ \textsf{ev:typecase2}$$

$$\frac{\begin{array}{c}\mathcal{L} \downarrow_* \varnothing \\ \theta = \{\varnothing \Rightarrow e_\varnothing,\ \{\} \Rightarrow e_{\{\}}, \cup \Rightarrow e_\cup,\ \mathcal{U} \Rightarrow e_\mathcal{U}\ \}\end{array}}{\mathcal{L}; \mathbf{setcase}\ \mathcal{L}\ \theta \mapsto \mathcal{L}; e_\varnothing}\ \textsf{ev:setcase-bot}$$

$$\frac{\begin{array}{c}\mathcal{L} \downarrow_* \{l\} \\ \theta = \{\varnothing \Rightarrow e_\varnothing,\ \{\} \Rightarrow e_{\{\}}, \cup \Rightarrow e_\cup,\ \mathcal{U} \Rightarrow e_\mathcal{U}\ \}\end{array}}{\mathcal{L}; \mathbf{setcase}\ \mathcal{L}\ \theta \mapsto \mathcal{L}; e_{\{\}}[\kappa][\hat{l}]}\ \textsf{ev:setcase-sing}$$

$$\frac{\begin{array}{c}\mathcal{L} \downarrow_* \mathcal{L}_1 \cup \mathcal{L}_2 \\ \theta = \{\varnothing \Rightarrow e_\varnothing,\ \{\} \Rightarrow e_{\{\}}, \cup \Rightarrow e_\cup,\ \mathcal{U} \Rightarrow e_\mathcal{U}\ \} \\ \mathcal{L}_1 \cup \mathcal{L}_2\ \mathit{norm}\end{array}}{\mathcal{L}; \mathbf{setcase}\ \mathcal{L}\ \theta \mapsto \mathcal{L}; e_\cup[\mathcal{L}_1][\mathcal{L}_2]}\ \textsf{ev:setcase-join}$$

$$\frac{\begin{array}{c}\mathcal{L} \downarrow_* \mathcal{U} \\ \theta = \{\varnothing \Rightarrow e_\varnothing,\ \{\} \Rightarrow e_{\{\}}, \cup \Rightarrow e_\cup,\ \mathcal{U} \Rightarrow e_\mathcal{U}\ \}\end{array}}{\mathcal{L}; \mathbf{setcase}\ \mathcal{L}\ \theta \mapsto \mathcal{L}; e_\mathcal{U}}\ \textsf{ev:setcase-top}$$

$$\frac{}{\mathcal{L}; \mathbf{lindex}\ \ell_i^\kappa \mapsto \mathcal{L}; i}\ \textsf{ev:lindex}$$

A.4.7 Congruence rules

$$\frac{\mathcal{L}; e_1 \mapsto \mathcal{L}'; e_1'}{\mathcal{L}; e_1 e_2 \mapsto \mathcal{L}'; e_1' e_2} \text{ ev:app-con1}$$

$$\frac{\mathcal{L}; e \mapsto \mathcal{L}'; e'}{\mathcal{L}; ve \mapsto \mathcal{L}'; ve'} \text{ ev:app-con2}$$

$$\frac{\mathcal{L}; e \mapsto \mathcal{L}'; e'}{\mathcal{L}; \{\!\{e\}\!\}^{\pm}_{l=\tau} \mapsto \mathcal{L}'; \{\!\{e'\}\!\}^{\pm}_{l=\tau}} \text{ ev:color-con}$$

$$\frac{\mathcal{L}; e \mapsto \mathcal{L}'; e'}{\mathcal{L}; \{\!\{e : \tau_1\}\!\}^{\pm}_{l=\tau_2} \mapsto \mathcal{L}'; \{\!\{e' : \tau_1\}\!\}^{\pm}_{l=\tau_2}} \text{ ev:hc-con}$$

$$\frac{\mathcal{L}; e_1 \mapsto \mathcal{L}'; e_1'}{\mathcal{L}; e_1 \bowtie e_2 \mapsto \mathcal{L}'; e_1' \bowtie e_2} \text{ ev:join-con1}$$

$$\frac{\mathcal{L}; e_2 \mapsto \mathcal{L}'; e_2'}{\mathcal{L}; v \bowtie e \mapsto \mathcal{L}'; v \bowtie e_2'} \text{ ev:join-con2}$$

$$\frac{\mathcal{L}; e \mapsto \mathcal{L}'; e'}{\mathcal{L}; \textbf{typecase}\ \tau\ e \mapsto \mathcal{L}'; \textbf{typecase}\ \tau\ e'} \text{ ev:typecase-con}$$

$$\frac{\mathcal{L}; e_1 \mapsto \mathcal{L}'; e_1'}{\mathcal{L}; e_1[\tau] \mapsto \mathcal{L}'; e_1'[\tau]} \text{ ev:tapp-con}$$

$$\frac{\mathcal{L}; e_1 \mapsto \mathcal{L}'; e_1'}{\mathcal{L}; e_1[\hat{l}] \mapsto \mathcal{L}'; e_1'[\hat{l}]} \text{ ev:lapp-con}$$

$$\frac{\mathcal{L}; e_1 \mapsto \mathcal{L}'; e_1'}{\mathcal{L}; e_1[\mathcal{L}] \mapsto \mathcal{L}'; e_1'[\mathcal{L}]} \text{ ev:sapp-con}$$

$$\frac{\mathcal{L}; e_1 \mapsto \mathcal{L}'; e_1'}{\mathcal{L}; e_1[\kappa] \mapsto \mathcal{L}'; e_1'[\kappa]} \text{ ev:kapp-con}$$