Unifying Nominal and Structural Ad-hoc Polymorphism (Extended Version)

Geoffrey Washburn Stephanie Weirich

March 5, 2004

Abstract

Ad-hoc polymorphism allows the execution of programs to depend on type information. In modern systems, it is useful for implementing generic operations over data structures, such as equality, marshalling, or traversal. In the past, there have been two different forms of ad-hoc polymorphism. The nominal form dispatches on the name of the type argument, whereas the structural form operates by decomposing the structure of types. In languages with user-defined types, these two approaches are very different. Operations defined by the nominal approach are "open"—they must be extended with specialized branches for user-defined types. In contrast, structurally defined operations are closed to extension. They automatically apply to user-defined types by treating them as their underlying definitions. Both approaches have their benefits, so it important to provide both capabilities in a language. Therefore we present an expressive language that supports both forms of ad-hoc polymorphism in a single framework. Among the language's features are the ability to define both "open" and "closed" operations with a single mechanism, the ability to naturally restrict the domain of type-analyzing operations, and new mechanisms for defining higher-order polytypism and manipulating generative type definitions.

1 Introduction

Ad-hoc polymorphism allows functions to alter their execution based on type information. Unlike parametric polymorphism, where the behavior of a polymorphic function is identical for all instances, with ad-hoc polymorphism the instance for the integer type may differ in execution radically from the instance for booleans. We call operations that depend on type information *polytypic*.

This form of polymorphism is a compelling addition to a typed programming language. It is well suited for dynamic environments—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 some of its most compelling applications, 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 various intermediate languages, and transformations implemented as traversals over these data structures. Without ad-hoc polymorphism, the same code for traversing abstract syntax must be implemented for each intermediate language. The generic traversals defined by ad-hoc polymorphism allows the programmer to concentrate on writing 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 the functionality provided by Haskell type classes [24]. The execution of an ad-hoc operation is determined solely by the name of the type argument (or the name of the head constructor, such as list.) It is difficult to call a polytypic operation defined in this manner a single "function" as it is composed of many disparate pieces of code. This nominal polytypism naturally limits the domain of an ad-hoc operation to those types where a definition has been provided. For example, a polymorphic addition function might be defined for integers and floating-point numbers, but not booleans. Polytypic operations defined in a nominal framework are naturally "open"; at any time they may be extended with instances for new types.

The second form of ad-hoc polymorphism is based on the *structural* analysis of types, such as the functionality provided by intensional type analysis [9]. Polytypic operations defined in this framework are defined by a case analysis of the various forms of type structure. Because they are defined by case analysis, these operations are naturally "closed" to extension. In fact, their operation must extend to all types at the point of definition.

In a language without user-defined types, these two approaches are roughly the same. However, many languages provide a mechanism, such Haskell's `newtype`s [20], for extending the language to include new forms of types. Although these new forms 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.

In the presence of user-defined types, neither purely nominal nor purely structural ad-hoc polymorphism is entirely satisfactory. Defining an operation in an open framework make it easy to extend to new user-defined types. However, it is difficult for the programmer to state with assurance that the operation is exhaustive. Furthermore, adding a new polytypic operation requires implementing cases for all existing types. Besides being fairly tedious, this way of defining polytypic operations leads to commingling of concerns, as the implementation becomes dispersed throughout a program.

Because closed operations cannot be extended to new application-specific types, structural systems treat a user-defined type as being equal to its definition. This approach destroys the distinctions that the user-defined types are designed to express. A polytypic operation cannot treat an "Age" in a different manner than a "PhoneNumber"—both are treated as integers. While some systems do allow ad-hoc definitions for user-defined types, there is a loss of abstraction—a polytypic operation can always determine a type's underlying representation.

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. The essence of $\lambda_{\mathcal{L}}$ is the following:

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 ad-hoc 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 polytypic operations with branches for these newly defined names. For this purpose, we introduce first-class type maps. Intuitively, these maps are branches for `typecase` that may be passed to polytypic operations, extending them to handle the new names. Also, $\lambda_{\mathcal{L}}$ includes support to easily coerce expressions mentioning new types to use the new type's underlying representations.

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 some sort of 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 and vital intermediate step towards improving the practicality of polytypic programming. In particular, this paper has the following contributions:

- We define a language that allows the definition of both "open" and "closed" polytypic operations. Previous work has chosen one or another framework, augmented with ad-hoc mechanisms to counter

$$
\begin{array}{rcl}
\mathsf{int} & \triangleq & \mathtt{l}_{\mathsf{int}} \\
\tau_1 \times \tau_2 & \triangleq & \mathtt{l}_{\times}\ \tau_1\ \tau_2 \\
\tau_1 \to \tau_2 & \triangleq & \mathtt{l}_{\to}\ \tau_1\ \tau_2 \\
\forall \alpha{:}\kappa \restriction \mathcal{L}.\tau & \triangleq & \mathtt{l}_{\forall}\ [\kappa]\ (\lambda \alpha{:}\kappa.\tau)\ \mathcal{L} \\
\forall^{+}\chi.\tau & \triangleq & \mathtt{l}_{\forall+}\ (\Lambda \chi.\tau) \\
\forall s.\tau & \triangleq & \mathtt{l}_{\forall\#}\ (\lambda s{:}\mathrm{LS}.\tau) \\
\mathcal{L} \Rightarrow \tau \restriction \mathcal{L}' & \triangleq & \mathtt{l}_{\mathsf{map}}\ \mathcal{L}\ \tau\ \mathcal{L}' \\
\forall^{*}\iota{:}\mathrm{L}(\kappa).\tau & \triangleq & \mathtt{l}_{\forall*}\ [\kappa]\ (\lambda \iota{:}\mathrm{L}(\kappa).\tau) \\
\tau'\langle \tau : \kappa \restriction \mathcal{L} \rangle & \triangleq & \mathtt{l}_{\mathsf{poly}}\ \tau'\ [\kappa]\ \tau\ \mathcal{L}
\end{array}
$$

Figure 1: Syntactic sugar for types

their difficulties.

- We define a language that allows programmers to statically restrict the domain of polytypic operations defined in a structural type system in a natural manner. Previous work [9, 4] requires that programmers use type-level intensional analysis or programming to makes such restrictions.
- We show how to reconcile `typecase` with the analysis of higher-order type constructors. Previous work [25] 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 underlying representations.

2 Programming in $\lambda_{\mathcal{L}}$

We begin by briefly describing the features of $\lambda_{\mathcal{L}}$ through examples. In Section 3 we will cover the semantics of these features in more detail.

Generative types At the core, $\lambda_{\mathcal{L}}$ is a functional programming language augmented with generative (also called branded) types and type constructors. New types are created with the expression $\mathtt{new}\ \iota{:}\kappa = \tau\ \mathtt{in}\ e$. This expression dynamically generates a new *label* (also called a name) equivalent to τ of kind κ and binds it to the label variable ι.

Labels are used as constants in the type language. For uniformity, all type constants (such as int and $\to$) of $\lambda_{\mathcal{L}}$ are represented by distinguished labels. For example, the type int is syntactic sugar for $\mathtt{l}_{\mathsf{int}}$. Figure 1 lists other forms of syntactic sugar for the types of $\lambda_{\mathcal{L}}$. We notate arbitrary label constants with $\mathtt{l}_i$'s.

The `new` expression statically introduces an isomorphism between ι and τ in the scope of e. Inside that scope, the operators $\{\!\{\cdot\}\!\}_{\iota}^{+}$ and $\{\!\{\cdot\}\!\}_{\iota}^{-}$ coerce expressions to and from that new type. For example, the following expression evaluates to 5.

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

Unlike Haskell `newtype`s, new type names in this language are created dynamically. Generating new names requires an operational effect at run time, but the term coercions are merely used for type checking and are not essential to execution. 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.

Type analysis with a restricted domain Polytypic operations are defined with `typecase`. This operator determines the head form of its type argument (such as int, $\times$, list, etc.) and selects the appropriate branch from a finite map from labels to expressions.

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

Figure 2: Polymorphic equality

For example, the expression $\texttt{typecase}\ \mathsf{int}\ \{\mathtt{l}_{\mathsf{int}} \Rightarrow 1, \mathtt{l}_{\mathsf{bool}} \Rightarrow 2\}$ evaluates to 1. A challenging part of the design of $\lambda_{\mathcal{L}}$ is ensuring that the argument to `typecase` is valid. It should never be a label that does not have a branch in the finite map. For example, we wish to rule out stuck expressions of the form $\texttt{typecase}\ \mathsf{bool}\ \{\mathtt{l}_{\mathsf{int}} \Rightarrow 2\}$.

For this reason, when checking a `typecase` expression, $\lambda_{\mathcal{L}}$ calculates the set of labels that may appear within the type argument to `typecase`, and requires that set be a subset of the set of labels that have branches in the finite map of the `typecase`. That way, progress is assured. So that the calculation of run-time labels is sound in the presence of polymorphism, we annotate a quantified type variable with the set of labels that may appear in types that instantiate it.

$$\Lambda\alpha{:}\star\restriction\mathtt{l}_{\mathsf{int}}\cup\mathtt{l}_{\mathsf{bool}}.\texttt{typecase}\ \alpha\ \{\mathtt{l}_{\mathsf{int}} \Rightarrow 2, \mathtt{l}_{\mathsf{bool}} \Rightarrow 3\}$$

In this case, we know that α must only be a type formed from the constructors int and bool, so the head form of α will have a match in the `typecase` expression. Alternatively, it is possible to specify the distinguished set of all labels ($\mathcal{U}$) as the annotation. In that case, types containing α are unanalyzable, because it is impossible to provide `typecase` branches for all possible labels. With $\mathcal{U}$, $\lambda_{\mathcal{L}}$ supports parametric polymorphism.

So far, the examples have only matched type constants of base kind $\star$. In general, the type of each branch in `typecase` is determined by the kind of the matched head label. After `typecase` determines the reified label in the head position of its argument, it steps to the corresponding map branch, applying any arguments that were applied to the head. For example, for arrow types: $\texttt{typecase}\ (\mathtt{l}_{\rightarrow}\ \tau_1\ \tau_2)\ \{\mathtt{l}_{\rightarrow} \Rightarrow e\} \mapsto e[\tau_1][\tau_2]$.

The function `eq` below implements a polymorphic equality function for data objects composed of integers, booleans, products and lists. In the following examples, let $\mathcal{L} = \mathtt{l}_{\mathsf{int}} \cup \mathtt{l}_{\mathsf{bool}} \cup \mathtt{l}_{\times} \cup \mathtt{l}_{\mathsf{list}}$.

In the case of products, $\mathtt{l}_{\times}$ has kind $\star \to \star \to \star$, so the branch abstracts two type variables for the subcomponents of the type. Likewise, in the $\mathtt{l}_{\mathsf{list}}$ case, the type of list elements is also abstracted. In the next section, we will make the relationship between the kind of the label and the type of the branch precise.

The ability to restrict the arguments of polytypic function in a natural way is valuable. For example, the polytypic equality function cannot be applied to values of function type. Therefore, SML [18] defines the set of "equality" types to be those that do not contain the $\mathtt{l}_{\rightarrow}$ constructor to ensure that each use of polymorphic equality is valid. Here, $\lambda_{\mathcal{L}}$ naturally makes this restriction by omitting $\{\mathtt{l}_{\rightarrow}\}$ from the set of labels for the argument of `eq`.

2.1 Reconciling generative types and type analysis

The function `eq` is closed to extension. However, with the creation of new names 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 leave `eq` as it is and at application, coerce all the arguments to `eq` so that their types do not contain new labels.
- We can rewrite `eq` to be extensible with new branches for the new type names.

2.1.1 Higher-order coercions

For the first scenario, we augment the calculus with an easy and efficient mechanism for coercing values with named types between their underlying representations and back. *Higher-order coercions* extend an isomorphism between l and τ to arbitrary types $\tau' l$ and $\tau' \tau$. Like first-order coercions, these operations have no run-time effect; they merely alter the types of expressions. While the semantics of these coercions is somewhat involved, this addition to the language is relatively straightforward.

For example, suppose we define a new type equivalent to a pair of integers with $\texttt{new}\ \iota{:}\star = \mathsf{int} \times \mathsf{int}$. We then create a value of this type, paired with a boolean using the expression $\texttt{let}\ x = \langle \{\!\{\langle 2, 2\rangle\}\!\}_\iota^+, \texttt{true}\rangle$. We cannot call $\mathsf{eq}[\iota \times \mathsf{bool}]\ x\ x$ because there is no match for the label ι in eq. However, we can use a higher-order coercion to change the type of x so that it does not include the type ι.

$$\mathsf{eq}[(\mathsf{int} \times \mathsf{int}) \times \mathsf{bool}] \{\!\{x : \lambda\beta{:}\star.\beta \times \mathsf{bool}\}\!\}_\iota^- \{\!\{x : \lambda\beta{:}\star.\beta \times \mathsf{bool}\}\!\}_\iota^-$$

There are two issues with relying on higher-order coercions to deal with new type names. The first is that it requires that the call site of a polytypic operation know the definition of every label that appears in the type. That way, the polytypic operation is not breaking any abstractions. Secondly, the programmer cannot distinguish labels from their underlying representations in the execution of a polytypic operation.

2.1.2 First-class maps

For the second scenario, we enhance the flexibility of the `typecase` operator, by making finite maps of branches for `typecase` *first-class*. Programmers may implement polytypic operations to allow new branches to be added at run time.

The type of a first-class map is $\mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L}_2$. The first component of this type is the domain of the map. The second and third components describe the types of these branches, as described in the next section. Using first-class maps, we can pass a branch for int's into the following operation (where $\bowtie$ joins together two maps):

$$\lambda x{:}\{\mathtt{l}_{\mathsf{int}}\} \Rightarrow (\lambda\alpha{:}\star.\mathsf{bool}) \restriction \{\mathtt{l}_{\mathsf{int}}\}.\texttt{typecase}\ \mathsf{int}\ (\{\mathtt{l}_{\mathsf{bool}} \Rightarrow \texttt{true}\} \bowtie x)$$

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 as the rightmost matching branch will be selected. In the following expression, if $\{\mathsf{int} \Rightarrow \texttt{false}\}$ is supplied for x, the expression will evaluate to `false`.

$$\lambda x{:}\{\mathtt{l}_{\mathsf{int}}\} \Rightarrow \lambda\alpha{:}\star.\mathsf{bool} \restriction \{\mathtt{l}_{\mathsf{int}}\}.\texttt{typecase}\ \mathsf{int}\ (\{\mathtt{l}_{\mathsf{bool}} \Rightarrow \texttt{false}, \mathtt{l}_{\mathsf{int}} \Rightarrow \texttt{true}\} \bowtie x)$$

Redefining the behavior of `typecase` for int may not be what the programmer intended, but allowing such scenarios does not affect the soundness of $\lambda_{\mathcal{L}}$. If the programmer wished to prevent this redefinition, she could join x on the left instead of the right. A more sophisticated language could include machinery to prevent shadowing from occurring.

2.1.3 Label and label set polymorphism

However, even though we may supply new branches to `typecase`, polytypic functions are still not extensible. We must specify which labels are in the domain of the map when it is passed to a polytypic function. Therefore, $\lambda_{\mathcal{L}}$ includes *label-set polymorphism*. A typical idiom for an extensible polytypic 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 polytypic functions that have been defined in this manner "open". For example, we can create an open version of eq as follows (again let $\mathcal{L} = \mathtt{l}_{\mathsf{int}} \cup \mathtt{l}_{\mathsf{bool}} \cup \mathtt{l}_{\times} \cup \mathtt{l}_{\mathsf{list}}$):

$$\begin{array}{l}
\Lambda s{:}\mathrm{Ls}.\lambda y{:}s \Rightarrow \lambda\alpha{:}\star.\alpha \to \alpha \to \mathsf{bool} \restriction s \cup \mathcal{L}. \\
\quad \texttt{fix}\ \mathsf{eq}{:}\forall\alpha{:}\star \restriction s \cup \mathcal{L}.\alpha \to \alpha \to \mathsf{bool}. \\
\quad\quad \Lambda\alpha{:}\star \restriction \mathcal{L}.\texttt{typecase}\ \alpha \\
\quad\quad\quad y \bowtie \{\mathtt{l}_{\mathsf{int}} \Rightarrow \ldots, \mathtt{l}_{\mathsf{bool}} \Rightarrow \ldots, \mathtt{l}_{\times} \Rightarrow \ldots, \mathtt{l}_{\mathsf{list}} \Rightarrow \ldots\}
\end{array}$$

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

Figure 3: Serialization

With this version of `eq`, we can treat generative types 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. We also cannot conclude that two objects with different labeled types are equal in the sense of `eq`, even though they may have the same underlying representation, because we cannot apply `eq` to them.

This calculus explicitly witnesses the design complexity of open polytypic operations. Suppose we wanted to call a polytypic operation, called `important` in the body of a polytypic serializer, called `tostring`. Intuitively, `important` should be called for each element of a pair to decide if recursion should continue. Because `tostring` can be applied to any type that provides a map for the new labels, `important` must also be applicable to all those types.

There are two solutions for this problem. The first is to supply the branches for `important` to `tostring`, as below. This solution is used by Dependency-Style Generic Haskell[17]. 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. In contrast, a *closed* polytypic operation may more easily call other *closed* polytypic functions.

2.2 Expressive polytypism

2.2.1 Analyzing label sets

An important property for a type-analyzing language is that it be fully reflexive. In other words, it should permit the analysis of all types that exist in the language. Because we have added two new type forms, restricted type quantification and maps, we must add machinery to $\lambda_{\mathcal{L}}$ to analyze these types. Both types contain label sets, so for full flexibility we add *label-set analysis*. Label-set analysis works similarly to `typecase`—it matches the outermost form of the label set (be it empty, a singleton label, the union of two sets, or the constant denoting the entire universe of labels.)

In the case that the label set is a singleton label, the programmer may want to find out which label it is. For this reason we also add *kind* and *label quantification* and *label analysis*. Labels have no interesting structure, so label analysis, performed with the keyword `lindex`, merely maps labels to integers.

For example, the following function computes a string representation of any label set.

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

2.2.2 Higher-order type analysis

In the polytypism of Hinze's system [10] (extended to run-time analysis by Weirich [25]), a polytypic operation is an interpretation of a type. 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 should produce the same results.

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 preserved at execution. However, even though `typecase` in this calculus only analyzes constructors of kind $\star$, we can still use $\lambda_{\mathcal{L}}$ to encode the instantiation of polytypic operators for constructors of kind $\kappa_1 \to \kappa_2$. Such *higher-order type analysis* is necessary to implement some polytypic operations.

For example, suppose f is an open polytypic operation of type $\forall s{:}\textsc{Ls}.\forall\beta{:}\star \restriction s.s \Rightarrow \tau' \restriction s \cup \mathcal{L} \to \tau'\beta$ and we want to use the instance of it for the type τ of kind $\star \to \star$, where $\mathcal{L}$ 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 of the constructor 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.

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

The above example is for types of kind $\star \to \star$. To apply f to types of other kinds, the kind of the analyzed type determines the expansion.

3 The $\lambda_{\mathcal{L}}$ language

Next we describe the semantics of $\lambda_{\mathcal{L}}$ in detail. The grammar for $\lambda_{\mathcal{L}}$ is in Figure 4. In this Section, so we only cover the most interesting aspects of the language. The complete semantics appears in the appendix.

The $\lambda_{\mathcal{L}}$ language includes label constants $\mathtt{l}^{\kappa}_{i}$, drawn from some countably infinite set. We assume there is some (unspecified) mechanism for determining the kind of label constants. When important, we annotate a constant with its kind. Label sets ($\mathcal{L}$) include the empty set ($\emptyset$), singletons, unions of sets, the complete set ($\mathcal{U}$), and label set variables. The judgments $\Delta \vDash l : \mathrm{L}(\kappa)$ and $\Delta \vDash \mathcal{L} : \textsc{Ls}$ determine when labels and label sets are well-formed within a given type context.

Types (also called constructors) include labels, variables, and several forms of abstraction. As well as type and kind abstractions, this language includes label and label set abstractions, which are given the kinds $\mathrm{L}(\kappa_1) \to \kappa_2$ and $\textsc{Ls} \to \kappa$ respectively.

To verify that the `typecase` analysis of a type has a matching branch, we conservatively determine the set of labels that could appear in the head position of the type. The well-formedness judgment for types, $\Delta \vdash \tau : \kappa \mid \mathcal{L}$, states that in the typing context Δ, the type constructor τ has kind κ and mentions labels in the set $\mathcal{L}$. In rules where the label set is unimportant, we elide it. The important rules for this judgment are those for label reification and variables.

$$\frac{\Delta \vDash l : \mathrm{L}(\kappa)}{\Delta \vdash l : \kappa \mid \{l\}} \qquad \frac{\vdash \Delta \qquad \alpha{:}\kappa \restriction \mathcal{L} \in \Delta}{\Delta \vdash \alpha : \kappa \mid \mathcal{L}} \qquad \frac{\vdash \Delta \qquad \alpha{:}\kappa \in \Delta}{\Delta \vdash \alpha : \kappa \mid \emptyset}$$

$$
\begin{array}{llll}
\textit{Kinds} & \kappa & ::= & \chi \mid \star \mid \kappa_1 \to \kappa_2 \mid \mathrm{L}(\kappa_1) \to \kappa_2 \mid \mathrm{Ls} \to \kappa \mid \forall \chi.\kappa \\
\textit{Labels} & l & ::= & \mathtt{l}_i^\kappa \mid \mathtt{l}_{\mathsf{int}} \mid \mathtt{l}_{\to} \mid \mathtt{l}_{\forall} \mid \mathtt{l}_{\forall^*} \mid \mathtt{l}_{\forall\#} \mid \mathtt{l}_{\forall^+} \mid \mathtt{l}_{\mathsf{map}} \mid \mathtt{l}_{\mathsf{poly}} \mid \iota \\
\textit{Label sets} & \mathcal{L} & ::= & \emptyset \mid \{l\} \mid \mathcal{L}_1 \cup \mathcal{L}_2 \mid \mathcal{U} \mid s \\
\textit{Types} & \sigma, \tau & ::= & \alpha \mid l \mid \lambda\alpha{:}\kappa.\tau \mid \tau_1\tau_2 \mid \lambda\iota{:}\mathrm{L}(\kappa).\tau \mid \tau \circ l \mid \lambda s{:}\mathrm{Ls}.\tau \mid \tau\mathcal{L} \mid \Lambda\chi.\tau \mid \tau[\kappa] \\
\textit{Terms} & e & ::= & x \mid i \mid \lambda x{:}\sigma.e \mid e_1 e_2 \mid \mathtt{fix}\ x{:}\sigma.e \mid \mathtt{new}\ \iota{:}\kappa = \tau\ \mathtt{in}\ e \mid \{\!\{e\}\!\}^{\pm}_{l=\tau} \mid \\
 & & & \{\!\{e : \tau\}\!\}^{\pm}_{l=\tau_2} \mid \mathtt{typecase}\ \tau\ e \mid \mathtt{setcase}\ \mathcal{L}\ e_\emptyset\ e_{\{\}}\ e_\cup\ e_\mathcal{U} \mid \\
 & & & \mathtt{lindex}\ l \mid \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{Type Contexts} & \Delta & ::= & \cdot \mid \Delta, \alpha{:}\kappa \mid \Delta, \iota{:}\mathrm{L}(\kappa) \mid \Delta, s{:}\mathrm{Ls} \mid \Delta, \alpha{:}\kappa \restriction \mathcal{L} \mid \Delta, \chi \\
\textit{Signatures} & \Sigma & ::= & \cdot \mid \Sigma, l{:}\kappa = \tau \\
\textit{Term Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\sigma \\
\textit{Type paths} & \rho & ::= & \bullet \mid \rho\ \tau \mid \rho \circ l \mid \rho\ \mathcal{L} \mid \rho\ [\kappa] \\
\textit{Term paths} & p & ::= & \bullet \mid p\ [\tau] \mid p\ [l]^* \mid p[\mathcal{L}] \mid p\ [\kappa]^+
\end{array}
$$

Figure 4: The $\lambda_\mathcal{L}$ language

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 abstraction is applied.

3.1 The term language

The semantics of the term language includes judgments for determining the well-formedness of a term $\Delta; \Gamma \vdash e : \sigma \mid \Sigma$ and the execution of a term in conjunction with a label set $\mathcal{L}; e \mapsto \mathcal{L}'; e'$. The first judgment 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. The second judgment 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* [19, 6]. The initial state of execution includes all label constants (such as $\mathtt{l}_{\mathsf{int}}$, $\mathtt{l}_{\to}$) 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. Below, we describe the dynamic and static semantics of label creation, coercions, first-class maps, and type, set and label abstraction and analysis.

3.1.1 Label creation

The dynamic and static semantics for `new` are:

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

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

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.

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

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

$$\frac{\tau' \Uparrow \lambda\alpha{:}\kappa.\mathcal{L}_1 \cup \mathcal{L}_2 \Rightarrow \tau' \restriction \mathcal{L}_3}{\mathcal{L}; \{\!\{v_1 \bowtie v_2 : \tau'\}\!\}^{\pm}_{l=\tau} \mapsto \mathcal{L}; \{\!\{v_1 : \lambda\alpha{:}\kappa.\mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L}_3\}\!\}^{\pm}_{l=\tau} \bowtie \{\!\{v_2 : \lambda\alpha{:}\kappa.\mathcal{L}_2 \Rightarrow \tau' \restriction \mathcal{L}_3\}\!\}^{\pm}_{l=\tau}}$$

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

Figure 5: Operational semantics for higher-order coercions (excerpt)

3.1.2 Coercions

To support isomorphisms between labels and types of *any* kind, the primitive coercions change the head constructor in the type of their arguments.

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

The syntax $\rho[\tau]$ denotes a type where τ is the head of the type path ρ. Operationally, the primitive coercion "out" cancels the primitive coercion "in".

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

Higher-order coercions extend the expressiveness of the primitive coercions to allow the non-head positions of a type to change.

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

Intuitively, a higher-order coercion "maps" the primitive coercions over an expression, guided by the type constructor τ'. If τ' is a path, then the higher-order coercion reduces to a primitive coercion. Otherwise, for each value form in $\lambda_{\mathcal{L}}$ there is an operational rule to map the coercion over that type (see Figure 5).

3.1.3 Type abstraction and analysis

With all the constraints that are enforced by the type system, polymorphism is essential to maintain flexibility. As such, $\lambda_{\mathcal{L}}$ provides not only type abstraction, but abstraction over kinds, label sets and labels. The static and dynamics semantics are standard, except for type application. At an application site, we check that the set of labels found in the argument type is contained within the set of labels accepted by the type abstraction:

$$\frac{\Delta; \Gamma \vdash e : \forall\alpha{:}\kappa \restriction \mathcal{L}.\sigma \mid \Sigma \qquad \Delta \vdash \tau : \kappa \mid \mathcal{L}' \qquad \Delta \vdash \mathcal{L}' \subseteq \mathcal{L}}{\Delta; \Gamma \vdash e[\tau] : \sigma[\tau/\alpha] \mid \Sigma}$$

The general rule for the execution of `typecase` is below:

$$\frac{\tau \Downarrow_* \rho[\mathtt{l}^{\kappa}_i] \qquad \{\mathtt{l}^{\kappa}_i \Rightarrow e'\} \in v \qquad \rho \rightsquigarrow p}{\mathcal{L}; \mathtt{typecase}\ \tau\ v \mapsto \mathcal{L}; p[e']}$$

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

Figure 6: Polykinded type equivalences

In this rule, `typecase` determines the weak-head normal form of its type argument, τ, obtaining some reified label $\mathtt{l}_i$ and a type path ρ. It chooses the rightmost matching branch from its map argument, v, and steps to the specified term, applying some series of type, label, label set, and kind 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 straightforward. Given some map e with domain $\mathcal{L}$ and a type argument τ that mentions labels in $\mathcal{L}'$, we check that the map can handle all possible labels in τ with $\mathcal{L}' \sqsubseteq \mathcal{L}$.

$$
\frac{\Delta;\Gamma \vdash e : \mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L}_2 \mid \Sigma \qquad \Delta \vdash \tau : \star \mid \mathcal{L}' \qquad \Delta \vdash \mathcal{L}_1 \subseteq \mathcal{L}_2 \qquad \Delta \vdash \mathcal{L}' \subseteq \mathcal{L}}{\Delta;\Gamma \vdash \mathtt{typecase}\ \tau\ e : \tau'\tau \mid \Sigma}
$$

Consider the rule for type checking a singleton map, below. 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.

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

For labels of higher kind, `typecase` will apply the matching branch to all of arguments in the path to the matched label. Therefore, the branch for that label must be able to accept all of those arguments. The correct type for this branch is determined by the kind of the label, with the notation $\tau'\langle\tau : \kappa \restriction \mathcal{L}\rangle$. The equality rules for this type appear in Figure 6. The label-set component of this kind-indexed type is used as the restriction for quantified type variables. The rule for `typecase` requires that both label sets in the type of the map be the same to ensure that it is safe to apply each branch to any subcomponents of the type argument.

3.1.4 Label set and label analysis

Because the language of label sets is fixed, the semantics of `setcase` explicitly has branches for all possible forms of label set.

$$
\frac{\begin{array}{ll}\Delta \vdash \tau' : \mathrm{Ls} \to \star \mid \cdot & \Delta;\Gamma \vdash e_\emptyset : \tau'\emptyset \mid \Sigma \qquad \Delta;\Gamma \vdash e_{\{\}} : \forall^+\chi.\forall^*\iota{:}\mathrm{L}(\chi).\tau'\{\iota\} \mid \Sigma \\ \Delta;\Gamma \vdash e_\cup : \forall s_1{:}\mathrm{Ls}.\forall s_1{:}\mathrm{Ls}.\tau'(s_1 \cup s_2) \mid \Sigma & \Delta;\Gamma \vdash e_\mathcal{U} : \tau'\mathcal{U} \mid \Sigma \qquad \Delta \vDash \mathcal{L} : \mathrm{Ls}\end{array}}{\Delta;\Gamma \vdash \mathtt{setcase}\ \mathcal{L}\ e_\emptyset\ e_{\{\}}\ e_\cup\ e_\mathcal{U} : \tau'\mathcal{L} \mid \Sigma}
$$

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. In contrast, the run-time analysis of labels is quite simple. Label analysis merely returns an integer, providing the programmer a way to distinguish between new labels.

$$
\frac{}{\mathcal{L};\mathtt{lindex}\ \mathtt{l}_i^\kappa \mapsto \mathcal{L};i}
$$

3.2 Properties

The $\lambda_{\mathcal{L}}$ language is type sound, following from the usual progress and preservation theorems [26]. The proofs of these theorems are straightforward inductions over derivations. We define the initial label set $\mathcal{L}_0$ to be $\mathtt{l}_{\mathsf{int}} \cup \mathtt{l}_{\rightarrow} \cup \mathtt{l}_{\forall} \cup \mathtt{l}_{\forall\#} \cup \mathtt{l}_{\forall^+} \cup \mathtt{l}_{\forall^*} \cup \mathtt{l}_{\mathsf{map}}$.

Theorem 3.1 (Progress) *If* $\cdot;\cdot \vdash e : \sigma \,|\, \Sigma$, *e is not a value,* $\cdot \vdash \mathcal{L}_0 \subseteq \mathcal{L}$, *and* $\cdot \vdash \mathrm{dom}(\Sigma) \subseteq \mathcal{L}$ *there exists some* $\mathcal{L}; e \mapsto \mathcal{L}'; e'$ *where* $\cdot \vdash \mathcal{L} \subseteq \mathcal{L}'$.

Theorem 3.2 (Preservation) *If* $\cdot;\cdot \vdash e : \sigma \,|\, \Sigma_1$, $\mathcal{L}; e \mapsto \mathcal{L}'; e'$, *where* $\cdot \vdash \mathcal{L}_0 \subseteq \mathcal{L}$, $\cdot \vdash \mathrm{dom}(\Sigma_1) \subseteq \mathcal{L}$ *then* $\cdot;\cdot \vdash e' : \sigma \,|\, \Sigma_2$ *where* $\cdot \vdash \mathrm{dom}(\Sigma_2) \subseteq \mathcal{L}'$ *and* $\cdot \vdash \Sigma_1 \leq \Sigma_2$.

We also conjecture that the coercions are not necessary to the operational semantics—that an untyped calculus where the coercions have been erased (preserving types, labels and label sets as terms for analysis) would have the same operational behavior.

4 Extensions

Default branches One way to increase expressiveness is to allow *default branches* that apply when no other branches match a label. To do so we add another form of map $\{\_ \Rightarrow e\}$ that is indexed by all labels. Because this branch can analyze any label, the domain of the map is $\mathcal{U}$. (With this extension, type variables restricted by $\mathcal{U}$ are no longer parametric.)

$$\frac{\Delta \vdash \tau' : \star \rightarrow \star \,|\, \cdot \qquad \Delta;\Gamma \vdash e : \forall^+\chi.\forall\alpha{:}\chi \restriction \mathcal{U}.\tau'\langle\alpha : \chi \restriction \mathcal{U}\rangle \,|\, \Sigma}{\Delta;\Gamma \vdash \{\_ \Rightarrow e\} : \mathcal{U} \Rightarrow \tau' \restriction \mathcal{U} \,|\, \Sigma}$$

This branch matches labels of *any* kind, so its type depends on the kind of the matched label. Therefore the type of this branch is a kind-polymorphic poly-kinded type.

Record and variant types Because $\lambda_{\mathcal{L}}$ deals with labels, with a small extension it can analyze types that deal with labels—records and variant types. There is *almost* enough machinery to represent these types in the type language. The missing component is support for finite maps from labels to types of kind $\star$. We can define the semantics of these finite type maps by analogy to label sets. Analysis of a label map is also analogous to label-set analysis. In the singleton case, both the label and its definition are provided to the branch.

Multi-place `typecase` Finally, we can extend $\lambda_{\mathcal{L}}$ to support multiplace `typecase`, allowing the definition of some of the most important polytypic operations such as map and zip. (See Hinze [10] for a more thorough explanation of the rôle of multiplace definitions.) To extend `typecase` to this form of polytypism, we must make several changes to $\lambda_{\mathcal{L}}$. The branches for `typecase`^{n} should have polykinded types indexed by n. Each version is a polykinded type based on n types of kind κ.

$$\frac{\Delta \vdash \tau_1 : \kappa \,|\, \mathcal{L}_1 \qquad \ldots \qquad \Delta \vdash \tau_n : \kappa \,|\, \mathcal{L}_n \qquad \Delta \vdash \tau' : \star^n \rightarrow \star \,|\, \mathcal{L}'}{\Delta \vdash \tau'\langle\tau_1 \ldots \tau_n : \kappa \restriction \mathcal{L}\rangle : \star \,|\, \mathcal{L}_1 \cup \ldots \mathcal{L}_n \cup \mathcal{L}'}$$

The key difference between the single polykinded type and the n-ary version is for function kinds. In that case, n type variables are abstracted.

$$\tau'\langle\tau_1 \ldots \tau_n : \kappa \rightarrow \kappa' \restriction \mathcal{L}\rangle = \forall\alpha_1{:}\kappa \restriction \mathcal{L}. \ldots \forall\alpha_n{:}\kappa \restriction \mathcal{L}.\tau'\langle(\tau_1\alpha_1) \ldots (\tau_n\alpha_n) : \kappa' \restriction \mathcal{L}\rangle$$

Because polykinded types are parameterized by n, map types must also come in different arities. To make sure that the calculus remains fully-reflexive, we must limit the number of different map types to a finite number.

To support higher-order analysis with the multiplace `typerec`, we allow new labels to be "isomorphic" to multiple types. The basic coercions, $\{\!\{e\}\!\}^{+i}_{l=\tau_1,\ldots,\tau_n}$ include an index specifying which definition should

be used. However, higher-order coercions can use all of the definitions at once. The annotations on the higher-order coercions abstract multiple variables, corresponding to the multiple definitions. For example, if ι is defined to be both τ_1 and τ_2 then $\{\!\{x : \lambda\alpha_1{:}\star.\lambda\alpha_2{:}\star.\mathtt{1}_{\rightarrow}\ \alpha_1\ \alpha_2\}\!\}^+_\iota$ coerces x from type $\tau_1 \rightarrow \tau_2$ to type $\iota \rightarrow \iota$.

5 Related work

There is much research on polytypic languages. *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 [9]. However, $\lambda_{\mathcal{L}}$ does not include a type-level analysis operator. Trifonov, Saha, and Shao extended Harper and Morrisett's language work to be fully-reflexive [23] by adding kind polymorphism. Weirich [25] extended run-time analysis to higher-order type constructors following the work of Hinze [10].

Generic programming uses the structure of datatypes to generate specialized operations at compile time. The Charity language [3] automatically generates folds for datatypes. PolyP [12] is an extension of Haskell that allows the definition of polytypic operations based on positive, regular datatypes. Functorial ML [14] bases polytypic operations on the composition of functors, and has lead to the programming language FISh [13]. Generic Haskell [2], following the work of Hinze [10] 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 [24] implement overloading by defining classes of types that have instances for a set of polytypic operations. Hinze and Peyton Jones [11] explored an extension to automatically derive type class instances by looking at the underlying structure of new types. Dependency-style Generic Haskell [17] 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 has not been provided.

Many languages use a form of generative types to represent application-specific abstractions. For example, Standard ML [18] and Haskell [20] 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 [8] or Stone and Harper [22]) 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 make sense because even if the definition is known, because 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 [6] source language dynamically checks predeclared subtyping relationships between type names. Lämmel and Peyton Jones [15] used dynamic type equality checks to implement a number of polytypic iterators. Rossberg's λ_N calculus [21] 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 slightly different semantics from ours. Higher-order coercions are reminiscent of the colored brackets of Grossman et al. [7], which are also used by Leifer et al. [16] to preserve type generativity when marshalling.

6 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; it may be worthwhile to consider extending $\lambda_{\mathcal{L}}$ 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 polytypic core of the language, we should extend it with type-level analysis of types. As shown in past work, it is impossible to assign types to some polytypic functions without this feature. One way to do so might be to extend the primitive-recursive operator of Trifonov et al. [23] to include first-class maps from labels to types.

Acknowledgments

Thanks to Steve Zdancewic, Dimitrios Vtiniotis, and Andreas Rossberg for helpful discussion.

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 theFourth 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 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 236–249. ACM Press, 2003.

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

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

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

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

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

[11] R. Hinze and S. P. 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.

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

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

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

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

[16] J. J. Leifer, G. Peskine, P. Sewell, and K. Wansbrough. Global abstraction-safe marshalling with hash types. In *Proc. 8th ICFP*, pages 87–98, Uppsala, Sweden, 2003.

[17] A. Löh, D. Clarke, and J. Jeuring. Dependency-style generic haskell. In *Proceedings of the eigth ACM SIGPLAN international conference on Functional programming*, pages 141–152. ACM Press, 2003.

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

[19] 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.

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

[21] 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. Extended version available from `http://www.ps.uni-sb.de/Papers/generativity-extended.html`.

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

[23] 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.

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

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

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

A Language

A.1 Syntax

Kinds	κ	::=	χ	*variables*
		|	$\star$	*types*
		|	$\kappa_1 \to \kappa_2$	*function kinds*
		|	$\mathrm{L}\mathrm{S} \to \kappa$	*label set functions*
		|	$\mathrm{L}(\kappa_1) \to \kappa_2$	*labels functions*
		|	$\forall\chi.\kappa$	*kind polymorphism*
Labels	l	::=	$\mathtt{l}_i^\kappa \mid \mathtt{l}_{\mathsf{int}} \mid \mathtt{l}_{\to} \mid \mathtt{l}_{\forall} \mid \mathtt{l}_{\forall^*} \mid \mathtt{l}_{\forall\#} \mid \mathtt{l}_{\forall^+} \mid \mathtt{l}_{\mathsf{map}} \mid \mathtt{l}_{\mathsf{poly}}$	*constants*
		|	ι	*variables*
Label sets	$\mathcal{L}$	::=	$\emptyset$	*empty*
		|	$\{l\}$	*singleton*
		|	s	*variable*
		|	$\mathcal{L}_1 \cup \mathcal{L}_2$	*union*
		|	$\mathcal{U}$	*universe*
Types	σ, τ	::=	$\alpha \mid \lambda\alpha{:}\kappa.\tau \mid \tau_1\tau_2$	*λ-calculus*
		|	$\lambda\iota{:}\mathrm{L}(\kappa).\tau \mid \tau \circ l$	*label abstraction*
		|	$\lambda s{:}\mathrm{L}\mathrm{S}.\tau \mid \tau\ \mathcal{L}$	*label set abstraction*
		|	$\Lambda\chi.\tau \mid \tau[\kappa]$	*kind abstraction*
		|	l	*label*
Terms	e	::=	$x \mid \lambda x{:}\sigma.e \mid e_1 e_2$	*λ-calculus*
		|	$\mathtt{fix}\ x{:}\sigma.e$	*recursion*
		|	i	*integers*
		|	$\mathtt{new}\ \iota{:}\kappa = \tau\ \mathtt{in}\ e$	*dynamic label creation*
		|	$\{\!\{e\}\!\}^{\pm}_{l=\tau}$	*primitive coercions*
		|	$\{\!\{e : \tau\}\!\}^{\pm}_{l=\tau_2}$	*extended coercions*
		|	$\mathtt{typecase}\ \tau\ e$	*type case analysis*
		|	$\mathtt{setcase}\ \mathcal{L}\ e_\emptyset\ e_{\{\}}\ e_\cup\ e_\mathcal{U}$	*label set case analysis*
		|	$\mathtt{lindex}\ l$	*label analysis*
		|	$\Lambda\alpha{:}\kappa \restriction \mathcal{L}.e \mid e[\tau]$	*constructor abstraction*
		|	$\Lambda^*\iota{:}\mathrm{L}(\kappa).e \mid e[l]^*$	*label abstraction*
		|	$\Lambda s{:}\mathrm{L}\mathrm{S}.e \mid e[\mathcal{L}]$	*label set abstraction*
		|	$\Lambda^+\chi.e \mid e[\kappa]^+$	*kind abstraction*
		|	$\varnothing$	*empty map*
		|	$\{l \Rightarrow e\}$	*singleton map*
		|	$e_1 \bowtie e_2$	*map join*
		|	$\{\_ \Rightarrow e\}$	*default map*

A.2 Static semantics

Type context well-formedness	$\vdash \Delta$
Kind well-formedness	$\Delta \vdash \kappa$
Label well-formedness	$\Delta \vDash l : \mathrm{L}(\kappa)$
Label set well-formedness	$\Delta \vDash \mathcal{L} : \mathrm{Ls}$
Signature well-formedness	$\Delta \vdash \Sigma$
Signature subsumption	$\Delta \vdash \Sigma_1 \leq \Sigma_2$
Label set subsumption	$\Delta \vdash \mathcal{L}_1 \subseteq \mathcal{L}_2$
Type well-formedness	$\Delta \vdash \tau : \kappa \,|\, \mathcal{L}$
Type equivalence	$\Delta \vdash \tau_1 = \tau_2 : \kappa$
Term context well-formedness	$\Delta \vdash \Gamma \,|\, \Sigma$
Term well-formedness	$\Delta; \Gamma \vdash e : \sigma \,|\, \Sigma$

Type Contexts	Δ	$::=$	$\cdot$	*empty context*
		$\mid$	$\Delta, \alpha{:}\kappa$	*λ-bound type variables*
		$\mid$	$\Delta, \iota{:}\mathrm{L}(\kappa)$	*label variables*
		$\mid$	$\Delta, s{:}\mathrm{Ls}$	*label set variables*
		$\mid$	$\Delta, \alpha{:}\kappa \upharpoonright \mathcal{L}$	*Λ-bound type variables*
		$\mid$	Δ, χ	*Kind variables*
Signatures	Σ	$::=$	$\cdot \mid \Sigma, l{:}\kappa = \tau$	
Term Contexts	Γ	$::=$	$\cdot \mid \Gamma, x{:}\sigma$	

Primitive labels and their kinds.

$\mathtt{l}_{\mathsf{int}}$	$\star$	
$\mathtt{l}_{\rightarrow}$	$\star \rightarrow \star \rightarrow \star$	*function type creator*
$\mathtt{l}_{\forall}$	$\forall\chi.(\chi \rightarrow \star) \rightarrow \mathrm{Ls} \rightarrow \star$	*type polymorphism*
$\mathtt{l}_{\forall^*}$	$\forall\chi.(\mathrm{L}(\chi) \rightarrow \star) \rightarrow \star$	*label polymorphism*
$\mathtt{l}_{\forall\#}$	$(\mathrm{Ls} \rightarrow \star) \rightarrow \star$	*label set polymorphism*
$\mathtt{l}_{\forall^+}$	$(\forall\chi.\star) \rightarrow \star$	*kind polymorphism*
$\mathtt{l}_{\mathsf{map}}$	$\mathrm{Ls} \rightarrow (\star \rightarrow \star) \rightarrow \star$	*map type*
$\mathtt{l}_{\mathsf{poly}}$	$(\star \rightarrow \star) \rightarrow \forall\chi.\chi \rightarrow \mathrm{Ls} \rightarrow \star$	*polykinded type*

A.2.1 Type context well-formedness

$$\mathsf{wfc{:}empty}\frac{}{\vdash \cdot} \qquad \mathsf{wfc{:}var}\frac{\Delta \vdash \kappa}{\vdash \Delta, \alpha{:}\kappa} \qquad \mathsf{wfc{:}lvar}\frac{\vdash \Delta \qquad \Delta \vdash \kappa}{\vdash \Delta, \iota{:}\mathrm{L}(\kappa)} \qquad \mathsf{wfc{:}svar}\frac{\vdash \Delta}{\vdash \Delta, s{:}\mathrm{Ls}}$$

$$\mathsf{wfc{:}kvar}\frac{\vdash \Delta}{\vdash \Delta, \chi} \qquad \mathsf{wfc{:}var\text{-}res}\frac{\Delta \vdash \kappa \qquad \Delta \vDash \mathcal{L} : \mathrm{Ls}}{\vdash \Delta, \alpha{:}\kappa \upharpoonright \mathcal{L}}$$

A.2.2 Kind well-formedness

$$\mathsf{wfk{:}var}\frac{\vdash \Delta \qquad \chi \in \Delta}{\Delta \vdash \chi} \qquad \mathsf{wfk{:}type}\frac{\vdash \Delta}{\Delta \vdash \star} \qquad \mathsf{wfk{:}larrow}\frac{\Delta \vdash \kappa_1 \qquad \Delta \vdash \kappa_2}{\Delta \vdash \mathrm{L}(\kappa_1) \rightarrow \kappa_2} \qquad \mathsf{wfk{:}sarrow}\frac{\Delta \vdash \kappa}{\Delta \vdash \mathrm{Ls} \rightarrow \kappa}$$

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

A.2.3 Label well-formedness

$$\mathsf{wfl{:}const}\frac{\vdash \Delta}{\Delta \vDash \mathtt{l}_i^{\kappa} : \mathrm{L}(\kappa)} \qquad \mathsf{wfl{:}var}\frac{\vdash \Delta \qquad \iota{:}\mathrm{L}(\kappa) \in \Delta}{\Delta \vDash \iota : \mathrm{L}(\kappa)}$$

A.2.4 Label set well-formedness

$$\textsf{wfls:empty}\frac{\vdash \Delta}{\Delta \vDash \emptyset : \textsc{Ls}} \qquad \textsf{wfls:sing}\frac{\Delta \vDash l : \mathrm{L}(\kappa)}{\Delta \vDash \{l\} : \textsc{Ls}} \qquad \textsf{wfls:var}\frac{\vdash \Delta \qquad s{:}\textsc{Ls} \in \Delta}{\Delta \vDash s : \textsc{Ls}}$$

$$\textsf{wfls:union}\frac{\Delta \vDash \mathcal{L}_1 : \textsc{Ls} \qquad \Delta \vDash \mathcal{L}_2 : \textsc{Ls}}{\Delta \vDash \mathcal{L}_1 \cup \mathcal{L}_2 : \textsc{Ls}} \qquad \textsf{wfls:univ}\frac{\vdash \Delta}{\Delta \vDash \mathcal{U} : \textsc{Ls}}$$

A.2.5 Signature well-formedness

$$\textsf{wfs:empty}\frac{\vdash \Delta}{\Delta \vdash \cdot} \qquad \textsf{wfs:def}\frac{\Delta \vdash \Sigma \qquad \Delta \vDash l : \mathrm{L}(\kappa) \qquad \Delta \vdash \tau : \kappa \,|\, \cdot}{\Delta \vdash \Sigma, l{:}\kappa = \tau}$$

A.2.6 Signature subsumption

$$\textsf{sigs:refl}\frac{\Delta \vdash \Sigma}{\Delta \vdash \Sigma \leq \Sigma} \qquad \textsf{sigs:def}\frac{\Delta \vdash \Sigma_1 \leq \Sigma_2 \qquad \Delta \vDash l : \mathrm{L}(\kappa) \qquad \Delta \vdash \tau : \kappa \,|\, \cdot}{\Delta \vdash \Sigma_1, l{:}\kappa = \tau \leq \Sigma_2, l{:}\kappa = \tau}$$

$$\textsf{sigs:def-ext}\frac{\Delta \vdash \Sigma_1 \leq \Sigma_2 \qquad \Delta \vDash l : \mathrm{L}(\kappa) \qquad \Delta \vdash \tau : \kappa \,|\, \cdot \qquad l \notin \Sigma_2}{\Delta \vdash \Sigma_1 \leq \Sigma_2, l{:}\kappa = \tau}$$

A.2.7 Type well-formedness

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

$$\textsf{twf:app}\frac{\Delta \vdash \tau_1 : \kappa_1 \to \kappa_2 \,|\, \mathcal{L}_1 \qquad \Delta \vdash \tau_2 : \kappa_1 \,|\, \mathcal{L}_2}{\Delta \vdash \tau_1 \tau_2 : \kappa_2 \,|\, \mathcal{L}_1 \cup \mathcal{L}_2} \qquad \textsf{twf:abs}\frac{\Delta, \alpha{:}\kappa_1 \vdash \tau : \kappa_2 \,|\, \mathcal{L} \qquad \Delta \vdash \kappa_1}{\Delta \vdash \lambda\alpha{:}\kappa_1.\tau : \kappa_1 \to \kappa_2 \,|\, \mathcal{L}}$$

$$\textsf{twf:ltype}\frac{\Delta \vDash l : \mathrm{L}(\kappa)}{\Delta \vdash l : \kappa \,|\, \{l\}} \qquad \textsf{twf:weak}\frac{\Delta \vdash \tau_1 : \kappa \,|\, \mathcal{L}_1 \qquad \Delta \vdash \mathcal{L}_1 \subseteq \mathcal{L}_2}{\Delta \vdash \tau_1 : \kappa \,|\, \mathcal{L}_2}$$

$$\textsf{twf:kabs}\frac{\Delta, \chi \vdash \tau : \kappa \,|\, \mathcal{L}}{\Delta \vdash \Lambda\chi.\tau : \forall\chi.\kappa \,|\, \mathcal{L}} \qquad \textsf{twf:kapp}\frac{\Delta \vdash \tau : \forall\chi.\kappa_2 \,|\, \mathcal{L} \qquad \Delta \vdash \kappa_1}{\Delta \vdash \tau[\kappa_1] : \kappa_2[\kappa_1/\chi] \,|\, \mathcal{L}}$$

$$\textsf{twf:labs}\frac{\Delta, \iota{:}\mathrm{L}(\kappa_1) \vdash \tau : \kappa_2 \,|\, \mathcal{L} \qquad \Delta \vdash \kappa_1 \qquad \iota \notin \mathcal{L}}{\Delta \vdash \lambda\iota{:}\mathrm{L}(\kappa_1).\tau : \mathrm{L}(\kappa_1) \to \kappa_2 \,|\, \mathcal{L}} \qquad \textsf{twf:lapp}\frac{\Delta \vdash \tau : \mathrm{L}(\kappa_1) \to \kappa_2 \,|\, \mathcal{L} \qquad \Delta \vDash l : \mathrm{L}(\kappa_1)}{\Delta \vdash \tau \circ l : \kappa_2 \,|\, \mathcal{L}}$$

$$\textsf{twf:sabs}\frac{\Delta, s{:}\textsc{Ls} \vdash \tau : \kappa \,|\, \mathcal{L} \qquad s \notin \mathcal{L}}{\Delta \vdash \lambda s{:}\textsc{Ls}.\tau : \textsc{Ls} \to \kappa \,|\, \mathcal{L}} \qquad \textsf{twf:sapp}\frac{\Delta \vdash \tau : \textsc{Ls} \to \kappa \,|\, \mathcal{L}_1 \qquad \Delta \vDash \mathcal{L}_2 : \textsc{Ls}}{\Delta \vdash \tau \mathcal{L}_2 : \kappa \,|\, \mathcal{L}_1}$$

A.2.8 Label set subsumption

$$\textsf{ss:refl}\frac{\Delta \vDash \mathcal{L} : \textsc{Ls}}{\Delta \vdash \mathcal{L} \subseteq \mathcal{L}} \qquad \textsf{ss:trans}\frac{\Delta \vdash \mathcal{L}_1 \subseteq \mathcal{L}_2 \qquad \Delta \vdash \mathcal{L}_2 \subseteq \mathcal{L}_3}{\Delta \vdash \mathcal{L}_1 \subseteq \mathcal{L}_3}$$

$$\textsf{ss:union-left}\frac{\Delta \vdash \mathcal{L}_1 \subseteq \mathcal{L} \qquad \Delta \vdash \mathcal{L}_2 \subseteq \mathcal{L}}{\Delta \vdash \mathcal{L}_1 \cup \mathcal{L}_2 \subseteq \mathcal{L}} \qquad \textsf{ss:union-right1}\frac{\Delta \vdash \mathcal{L} \subseteq \mathcal{L}_1 \qquad \Delta \vDash \mathcal{L}_2 : \textsc{Ls}}{\Delta \vdash \mathcal{L} \subseteq \mathcal{L}_1 \cup \mathcal{L}_2}$$

$$\textsf{ss:union-right2}\frac{\Delta \vdash \mathcal{L} \subseteq \mathcal{L}_2 \qquad \Delta \vDash \mathcal{L}_1 : \textsc{Ls}}{\Delta \vdash \mathcal{L} \subseteq \mathcal{L}_1 \cup \mathcal{L}_2} \qquad \textsf{ss:empty}\frac{\Delta \vDash \mathcal{L} : \textsc{Ls}}{\Delta \vdash \emptyset \subseteq \mathcal{L}} \qquad \textsf{ss:univ}\frac{\Delta \vDash \mathcal{L} : \textsc{Ls}}{\Delta \vdash \mathcal{L} \subseteq \mathcal{U}}$$

A.2.9 Label set equivalence

$$\textsf{seq:deriv}\frac{\Delta \vdash \mathcal{L}_1 \subseteq \mathcal{L}_2 \qquad \Delta \vdash \mathcal{L}_2 \subseteq \mathcal{L}_1}{\Delta \vDash \mathcal{L}_1 = \mathcal{L}_2}$$

A.2.10 Type equivalence

$$\textsf{teq:refl}\frac{\Delta \vdash \tau : \kappa \,|\, \cdot}{\Delta \vdash \tau = \tau : \kappa} \qquad \textsf{teq:sym}\frac{\Delta \vdash \tau_1 = \tau_2 : \kappa}{\Delta \vdash \tau_2 = \tau_1 : \kappa} \qquad \textsf{teq:trans}\frac{\Delta \vdash \tau_1 = \tau_2 : \kappa \qquad \Delta \vdash \tau_2 = \tau_3 : \kappa}{\Delta \vdash \tau_1 = \tau_3 : \kappa}$$

$$\textsf{teq:abs-beta}\frac{\Delta \vdash \lambda\alpha{:}\kappa_1.\tau_1 : \kappa_1 \to \kappa_2 \,|\, \cdot \qquad \Delta \vdash \tau_2 : \kappa_1 \,|\, \cdot}{\Delta \vdash (\lambda\alpha{:}\kappa_1.\tau_1)\tau_2 = \tau_1[\tau_2/\alpha] : \kappa_2} \qquad \textsf{teq:abs-eta}\frac{\Delta \vdash \tau : \kappa_1 \to \kappa_2 \,|\, \cdot}{\Delta \vdash \lambda\alpha{:}\kappa_1.\tau\alpha = \tau : \kappa_1 \to \kappa_2}$$

$$\textsf{teq:app-con}\frac{\Delta \vdash \tau_1 = \tau_3 : \kappa_1 \to \kappa_2 \qquad \Delta \vdash \tau_2 = \tau_4 : \kappa_1}{\Delta \vdash \tau_1\tau_2 = \tau_3\tau_4 : \kappa_2} \qquad \textsf{teq:abs-con}\frac{\Delta, \alpha{:}\kappa_1 \vdash \tau_1 = \tau_2 : \kappa_2 \qquad \Delta \vdash \kappa_1}{\Delta \vdash \lambda\alpha{:}\kappa_1.\tau_1 = \lambda\alpha{:}\kappa_1.\tau_2 : \kappa_1 \to \kappa_2}$$

$$\textsf{teq:kapp-con}\frac{\Delta \vdash \tau_1 = \tau_2 : \forall\chi.\kappa_2 \qquad \Delta \vdash \kappa_1}{\Delta \vdash \tau_1[\kappa_1] = \tau_2[\kappa_1] : \kappa_2[\kappa_1/\chi]} \qquad \textsf{teq:kabs-con}\frac{\Delta, \chi \vdash \tau_1 = \tau_2 : \kappa_2}{\Delta \vdash \Lambda\chi.\tau_1 = \Lambda\chi.\tau_2 : \kappa_1 \to \kappa_2}$$

$$\textsf{teq:kabs-beta}\frac{\Delta \vdash \Lambda\chi.\tau_1 : \forall\chi.\kappa_2 \,|\, \cdot \qquad \Delta \vdash \kappa_1}{\Delta \vdash (\Lambda\chi.\tau_1)[\kappa_1] = \tau_2[\kappa_1/\chi] : \kappa_2[\kappa_1/\chi]} \qquad \textsf{teq:kabs-eta}\frac{\Delta \vdash \tau : \forall\chi.\kappa \,|\, \cdot}{\Delta \vdash \Lambda\chi.\tau[\chi] = \tau : \forall\chi.\kappa}$$

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

$$\textsf{teq:labs-con}\frac{\Delta, \iota{:}\mathrm{L}(\kappa_1) \vdash \tau_1 = \tau_2 : \kappa_2}{\Delta \vdash \lambda\iota{:}\mathrm{L}(\kappa_1).\tau_1 = \lambda\iota{:}\mathrm{L}(\kappa_1).\tau_2 : \mathrm{L}(k_1) \to \kappa_2}$$

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

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

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

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

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

$$\textsf{teq:polyk-arrow}\frac{\Delta \vdash \tau : \kappa_1 \to \kappa_2 \,|\, \cdot \qquad \Delta \vdash \tau' : \star \to \star \,|\, \cdot \qquad \Delta \vDash \mathcal{L} : \mathrm{Ls}}{\Delta \vdash \tau' \langle \tau : \kappa_1 \to \kappa_2) \restriction \mathcal{L} \rangle = \forall \alpha{:}\kappa \restriction \mathcal{L}.\tau' \langle \tau \alpha : \kappa_2 \restriction \mathcal{L} \rangle : \star}$$

$$\textsf{teq:polyk-larrow}\frac{\Delta \vdash \tau : \mathrm{L}(\kappa_1) \to \kappa_2 \,|\, \cdot \qquad \Delta \vdash \tau' : \star \to \star \,|\, \cdot \qquad \Delta \vDash \mathcal{L} : \mathrm{Ls}}{\Delta \vdash \tau' \langle \tau : \mathrm{L}(\kappa_1) \to \kappa_2 \restriction \mathcal{L} \rangle = \forall^* \iota{:}\mathrm{L}(\kappa_1).\tau' \langle \tau \iota : \kappa_2 \restriction \mathcal{L} \rangle : \star}$$

$$\textsf{teq:polyk-sarrow}\frac{\Delta \vdash \tau : \mathrm{Ls} \to \kappa \,|\, \cdot \qquad \Delta \vdash \tau' : \star \to \star \,|\, \cdot \qquad \Delta \vDash \mathcal{L} : \mathrm{Ls}}{\Delta \vdash \tau' \langle \tau : \mathrm{Ls} \to \kappa \restriction \mathcal{L} \rangle = \forall s{:}\mathrm{Ls}.\tau' \langle \tau s : \kappa \restriction \mathcal{L} \rangle : \star}$$

$$\textsf{teq:polyk-all}\frac{\Delta \vdash \tau : \forall \chi.\kappa \,|\, \cdot \qquad \Delta \vdash \tau' : \star \to \star \,|\, \cdot \qquad \Delta \vDash \mathcal{L} : \mathrm{Ls}}{\Delta \vdash \tau' \langle \tau : \forall \chi.\kappa \restriction \mathcal{L} \rangle = \forall^+ \chi.\tau' \langle \tau[\chi] : \kappa \restriction \mathcal{L} \rangle : \star}$$

A.2.11 Term context well-formedness

$$\textsf{wftc:empty}\frac{\Delta \vdash \Sigma}{\Delta \vdash \cdot \,|\, \Sigma} \qquad \textsf{wftc:var}\frac{\Delta \vdash \Gamma \,|\, \Sigma \qquad \Delta \vdash \sigma : \star \,|\, \cdot}{\Delta \vdash \Gamma, x{:}\sigma \,|\, \Sigma}$$

A.2.12 Term well-formedness

$$\textsf{wf:var}\frac{\Delta \vdash \Gamma \,|\, \Sigma \qquad x{:}\sigma \in \Gamma}{\Delta; \Gamma \vdash x : \sigma \,|\, \Sigma} \qquad \textsf{wf:int}\frac{\Delta \vdash \Gamma \,|\, \Sigma}{\Delta; \Gamma \vdash i : \mathsf{int} \,|\, \Sigma} \qquad \textsf{wf:abs}\frac{\Delta; \Gamma, x{:}\sigma_1 \vdash e : \sigma_2 \,|\, \Sigma}{\Delta; \Gamma \vdash \lambda x{:}\sigma_1.e : \sigma_1 \to \sigma_2 \,|\, \Sigma}$$

$$\textsf{wf:app}\frac{\Delta; \Gamma \vdash e_1 : \sigma_1 \to \sigma_2 \,|\, \Sigma \qquad \Delta; \Gamma \vdash e_2 : \sigma_1 \,|\, \Sigma}{\Delta; \Gamma \vdash e_1 e_2 : \sigma_2 \,|\, \Sigma} \qquad \textsf{wf:fix}\frac{\Delta; \Gamma, x{:}\sigma \vdash e : \sigma \,|\, \Sigma}{\Delta; \Gamma \vdash \mathtt{fix}\ x{:}\sigma.e : \sigma \,|\, \Sigma}$$

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

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

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

$$\textsf{wf:weak}\frac{\Delta; \Gamma \vdash e : \sigma_1 \,|\, \Sigma_1 \qquad \Delta \vdash \sigma_1 = \sigma_2 : \star \qquad \Delta \vdash \Sigma_1 \leq \Sigma_2}{\Delta; \Gamma \vdash e : \sigma_2 \,|\, \Sigma_2}$$

$$\textsf{wf:env-empty}\frac{\Delta \vDash \mathcal{L} : \mathrm{Ls} \qquad \Delta \vdash \Gamma \,|\, \Sigma \qquad \Delta \vdash \tau' : \star \to \star \,|\, \cdot}{\Delta; \Gamma \vdash \varnothing : \emptyset \Rightarrow \tau' \restriction \mathcal{L} | \, \Sigma}$$

$$\textsf{wf:env-branch}\frac{\Delta \vDash \mathcal{L} : \mathrm{Ls} \qquad \Delta \vdash \tau' : \star \to \star \,|\, \cdot \qquad \Delta \vDash l : \mathrm{L}(\kappa) \qquad \Delta; \Gamma \vdash e : \tau' \langle l : \kappa \restriction \mathcal{L} \rangle \,|\, \Sigma}{\Delta; \Gamma \vdash \{l \Rightarrow e\} : \{l\} \Rightarrow \tau' \restriction \mathcal{L} \,|\, \Sigma}$$

$$\textsf{wf:env-else}\frac{\Delta \vdash \tau' : \star \to \star \,|\, \cdot \qquad \Delta; \Gamma \vdash e : \forall^+ \chi.\forall \alpha{:}\chi \restriction \mathcal{U}.\tau' \langle \alpha : \chi \restriction \mathcal{U} \rangle \,|\, \Sigma}{\Delta; \Gamma \vdash \{\_ \Rightarrow e\} : \mathcal{U} \Rightarrow \tau' \restriction \mathcal{U} | \, \Sigma}$$

$$\textsf{wf:env-join}\frac{\Delta; \Gamma \vdash e_1 : \mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L} \,|\, \Sigma \qquad \Delta; \Gamma \vdash e_2 : \mathcal{L}_2 \Rightarrow \tau' \restriction \mathcal{L} \,|\, \Sigma}{\Delta; \Gamma \vdash e_1 \bowtie e_2 : \mathcal{L}_1 \cup \mathcal{L}_2 \Rightarrow \tau' \restriction \mathcal{L} \,|\, \Sigma}$$

$$\textsf{wf:typecase}\frac{\Delta;\Gamma \vdash e : \mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L}_2 \mid \Sigma \qquad \Delta \vdash \tau : \star \mid \mathcal{L}' \qquad \Delta \vdash \mathcal{L}_1 \subseteq \mathcal{L}_2 \qquad \Delta \vdash \mathcal{L}' \subseteq \mathcal{L}}{\Delta;\Gamma \vdash \texttt{typecase}\ \tau\ e : \tau'\tau \mid \Sigma}$$

$$\textsf{wf:setcase}\frac{\begin{array}{c}\Delta \vdash \tau' : \mathrm{Ls} \to \star \mid \cdot \qquad \Delta;\Gamma \vdash e_\emptyset : \tau'\emptyset \mid \Sigma \qquad \Delta;\Gamma \vdash e_{\{\}} : \forall^+\chi.\forall^*\iota{:}\mathrm{L}(\chi).\tau'\{\iota\} \mid \Sigma \\ \Delta;\Gamma \vdash e_\cup : \forall s_1{:}\mathrm{Ls}.\forall s_1{:}\mathrm{Ls}.\tau'(s_1 \cup s_2) \mid \Sigma \qquad \Delta;\Gamma \vdash e_\mathcal{U} : \tau'\mathcal{U} \mid \Sigma \qquad \Delta \vDash \mathcal{L} : \mathrm{Ls}\end{array}}{\Delta;\Gamma \vdash \texttt{setcase}\ \mathcal{L}\ e_\emptyset\ e_{\{\}}\ e_\cup\ e_\mathcal{U} : \tau'\mathcal{L} \mid \Sigma}$$

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

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

$$\textsf{wf:tapp}\frac{\Delta;\Gamma \vdash e : \forall\alpha{:}\kappa \restriction \mathcal{L}.\sigma \mid \Sigma \qquad \Delta \vdash \tau : \kappa \mid \mathcal{L}' \qquad \Delta \vdash \mathcal{L}' \subseteq \mathcal{L}}{\Delta;\Gamma \vdash e[\tau] : \sigma[\tau/\alpha] \mid \Sigma}$$

$$\textsf{wf:labs}\frac{\Delta, \iota{:}\mathrm{L}(\kappa);\Gamma \vdash e : \sigma \mid \Sigma \qquad \Delta \vdash \kappa \qquad \iota \notin \Sigma}{\Delta;\Gamma \vdash \Lambda^*\iota{:}\mathrm{L}(\kappa).e : \forall^*\iota{:}\mathrm{L}(\kappa).\sigma \mid \Sigma} \qquad \textsf{wf:lapp}\frac{\Delta;\Gamma \vdash e : \forall\iota{:}\mathrm{L}(\kappa).\sigma \mid \Sigma \qquad \Delta \vDash l : \mathrm{L}(\kappa)}{\Delta;\Gamma \vdash e[l]^* : \sigma[l/\iota] \mid \Sigma}$$

$$\textsf{wf:sabs}\frac{\Delta, s{:}\mathrm{Ls};\Gamma \vdash e : \sigma \mid \Sigma \qquad s \notin \Sigma}{\Delta;\Gamma \vdash \Lambda s{:}\mathrm{Ls}.e : \forall s.\sigma \mid \Sigma} \qquad \textsf{wf:sapp}\frac{\Delta;\Gamma \vdash e : \forall s{:}\mathrm{Ls}.\sigma \mid \Sigma \qquad \Delta \vDash \mathcal{L} : \mathrm{Ls}}{\Delta;\Gamma \vdash e[\mathcal{L}] : \sigma[\mathcal{L}/s] \mid \Sigma}$$

$$\textsf{wf:kabs}\frac{\Delta, \chi;\Gamma \vdash e : \sigma \mid \Sigma}{\Delta;\Gamma \vdash \Lambda^+\chi.e : \forall^+\chi.\sigma \mid \Sigma} \qquad \textsf{wf:kapp}\frac{\Delta;\Gamma \vdash e : \forall\chi.\sigma \mid \Sigma \qquad \Delta \vdash \kappa}{\Delta;\Gamma \vdash e[\kappa]^+ : \sigma[\kappa/\alpha] \mid \Sigma}$$

A.3 Dynamic semantics

Small-step evaluation	$\mathcal{L}; e \mapsto \mathcal{L}'; e'$
Weak-head reduction	$\tau \Downarrow \tau'$
Constructor eta-expansion	$\tau \Uparrow \tau'$
Label set reduction	$\mathcal{L}_1 \downarrow \mathcal{L}_2$
Path conversion	$\rho \rightsquigarrow p$

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

A.3.1 Weak-head reduction for types

$$\textsf{whr:abs-beta}\frac{}{(\lambda\alpha{:}\kappa.\tau_1)\tau_2 \Downarrow \tau_1[\tau_2/\alpha]} \qquad \textsf{whr:abs-con}\frac{\tau \Downarrow \tau'}{\lambda\alpha{:}\kappa.\tau \Downarrow \lambda\alpha{:}\kappa.\tau'} \qquad \textsf{whr:app-con}\frac{\tau_1 \Downarrow \tau_1'}{\tau_1\tau_2 \Downarrow \tau_1'\tau_2}$$

$$\textsf{whr:labs-beta}\frac{}{(\lambda\iota{:}\mathrm{L}(\kappa).\tau)l \Downarrow \tau[l/\iota]} \qquad \textsf{whr:lapp-con}\frac{\tau \Downarrow \tau'}{\tau \circ l \Downarrow \tau' \circ l} \qquad \textsf{whr:sabs-beta}\frac{}{(\lambda s{:}\mathrm{Ls}.\tau)\mathcal{L} \Downarrow \tau[\mathcal{L}/s]}$$

$$\textsf{whr:sapp-con}\frac{\tau \Downarrow \tau'}{\tau\mathcal{L} \Downarrow \tau'\mathcal{L}} \qquad \textsf{whr:kabs-beta}\frac{}{(\Lambda\chi.\tau)[\kappa] \Downarrow \tau[\kappa/\chi]} \qquad \textsf{whr:kapp-con}\frac{\tau \Downarrow \tau'}{\tau[\kappa] \Downarrow \tau'[\kappa]}$$

$$\textsf{whr:polyk-type}\frac{}{\tau'\langle\tau : \star \restriction \mathcal{L}\rangle \Downarrow \tau'\tau} \qquad \textsf{whr:polyk-arrow}\frac{}{\tau'\langle\tau : \kappa_1 \to \kappa_2 \restriction \mathcal{L}\rangle \Downarrow \forall\alpha{:}\kappa_1 \restriction \mathcal{L}.\tau'\langle\tau\alpha : \kappa_2 \restriction \mathcal{L}\rangle}$$

$$\textsf{whr:polyk-larrow}\frac{}{\tau'\langle\tau : \mathrm{L}(\kappa_1) \to \kappa_2 \restriction \mathcal{L}\rangle \Downarrow \forall^*\iota{:}\mathrm{L}(\kappa_1).\tau'\langle\tau\iota : \kappa_2 \restriction \mathcal{L}\rangle}$$

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

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

A.3.2 Constructor eta-expansion

$$\textsf{tcee:tycon}\frac{\tau\ \alpha \Downarrow_* \tau'}{\tau \Uparrow \lambda\alpha{:}\star.\tau'}$$

A.3.3 Reduction for label sets

$$\textsf{lsr:union-con1}\frac{\mathcal{L}_1 \downarrow \mathcal{L}_1'}{\mathcal{L}_1 \cup \mathcal{L}_2 \downarrow \mathcal{L}_1' \cup \mathcal{L}_2} \qquad \textsf{lsr:union-con2}\frac{\mathcal{L}_2 \downarrow \mathcal{L}_2'}{\mathcal{L}_1 \cup \mathcal{L}_2 \downarrow \mathcal{L}_1 \cup \mathcal{L}_2'} \qquad \textsf{lsr:union-empty1}\frac{}{\emptyset \cup \mathcal{L} \downarrow \mathcal{L}}$$

$$\textsf{lsr:union-empty2}\frac{}{\mathcal{L} \cup \emptyset \downarrow \mathcal{L}} \qquad \textsf{lsr:union-univ1}\frac{}{\mathcal{U} \cup \mathcal{L} \downarrow \mathcal{U}} \qquad \textsf{lsr:union-univ2}\frac{}{\mathcal{L} \cup \mathcal{U} \downarrow \mathcal{U}}$$

$$\textsf{lsr:union-swap}\frac{\forall l_j \sqsubseteq \mathcal{L} \qquad i < j}{\mathcal{L} \cup l_i \downarrow l_i \cup \mathcal{L}} \qquad \textsf{lsr:union-comm}\frac{}{(\mathcal{L}_1 \cup \mathcal{L}_2) \cup \mathcal{L}_3 \downarrow \mathcal{L}_1 \cup (\mathcal{L}_2 \cup \mathcal{L}_3)}$$

A.3.4 Path conversion

$$\textsf{pc:hole}\frac{}{\bullet \rightsquigarrow \bullet} \qquad \textsf{pc:app}\frac{\rho \rightsquigarrow p}{\rho\ \tau \rightsquigarrow p\ [\tau]} \qquad \textsf{pc:lapp}\frac{\rho \rightsquigarrow p}{\rho\ \circ l \rightsquigarrow p\ [l]^*} \qquad \textsf{pc:sapp}\frac{\rho \rightsquigarrow p}{\rho\ \mathcal{L} \rightsquigarrow p\ [\mathcal{L}]}$$

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

A.3.5 Computation rules

$$\textsf{ev:abs-beta}\ \frac{}{\mathcal{L};(\lambda x{:}\sigma.e_1)e_2 \mapsto \mathcal{L};e_1[e_2/x]} \qquad \textsf{ev:fix-beta}\ \frac{}{\mathcal{L};\texttt{fix}\ x{:}\sigma.e \mapsto \mathcal{L};e[\texttt{fix}\ x{:}\sigma.e/x]}$$

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

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

$$\textsf{ev:in-out}\ \frac{}{\mathcal{L};\{\!\{\{\!\{v\}\!\}^+_{l=\tau}\}\!\}^-_{l=\tau} \mapsto \mathcal{L};v} \qquad \textsf{ev:new}\ \frac{\mathtt{l}^\kappa_i \notin \mathcal{L}}{\mathcal{L};\texttt{new}\ \iota{:}\kappa = \tau\ \texttt{in}\ e \mapsto \mathcal{L}\cup\{\mathtt{l}^\kappa_i\};e[\mathtt{l}^\kappa_i/\iota]}$$

$$\textsf{ev:hcolor-base}\ \frac{\tau' \Uparrow \lambda\alpha{:}\kappa.\rho[\alpha]}{\mathcal{L};\{\!\{v:\tau'\}\!\}^\pm_{l=\tau} \mapsto \mathcal{L};\{\!\{\{\!\{v:\lambda\alpha{:}\kappa.\rho[\tau]\}\!\}^\pm_{l=\tau}\}\!\}^\pm_{l=\tau}} \qquad \textsf{ev:hcolor-int}\ \frac{\tau' \Uparrow \lambda\alpha{:}\kappa.\mathsf{int}}{\mathcal{L};\{\!\{i:\tau'\}\!\}^\pm_{l=\tau} \mapsto \mathcal{L};i}$$

$$\textsf{ev:hcolor-abs1}\ \frac{\tau' \Uparrow \lambda\alpha{:}\kappa.\tau_1 \to \tau_2}{\mathcal{L};\{\!\{\lambda x{:}\tau_1.e:\tau'\}\!\}^+_{l=\tau} \mapsto \mathcal{L};\lambda x{:}(\tau_1[l/\alpha]).\{\!\{e[\{\!\{x:\lambda\alpha{:}\kappa.\tau_1\}\!\}^-_{l=\tau}/x]:\lambda\alpha{:}\kappa.\tau_2\}\!\}^+_{l=\tau}}$$

$$\textsf{ev:hcolor-abs2}\ \frac{\tau' \Uparrow \lambda\alpha{:}\kappa.\tau_1 \to \tau_2}{\mathcal{L};\{\!\{\lambda x{:}\tau_1.e:\tau'\}\!\}^-_{l=\tau} \mapsto \mathcal{L};\lambda x{:}(\tau_1[\tau/\alpha]).\{\!\{e[\{\!\{x:\lambda\alpha{:}\kappa.\tau_1\}\!\}^+_{l=\tau}/x]:\lambda\alpha{:}\kappa.\tau_2\}\!\}^-_{l=\tau}}$$

$$\textsf{ev:hcolor-empty}\ \frac{\tau' \Uparrow \lambda\alpha{:}\kappa.\mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L}_2}{\mathcal{L};\{\!\{\varnothing:\tau'\}\!\}^\pm_{l=\tau} \mapsto \mathcal{L};\varnothing}$$

$$\textsf{ev:hcolor-sing}\ \frac{\tau' \Uparrow \lambda\alpha{:}\kappa.\mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L}_2}{\mathcal{L};\{\!\{\{\mathtt{l}^\kappa_i \Rightarrow e'\}:\tau'\}\!\}^\pm_{l=\tau} \mapsto \mathcal{L};\{\mathtt{l}^k_i \Rightarrow \{\!\{e':\lambda\alpha{:}\star.\tau'\langle\mathtt{l}^k_i:\kappa \restriction \mathcal{L}_2\rangle\}\!\}^\pm_{l=\tau}\}}$$

$$\textsf{ev:hcolor-join}\ \frac{\tau' \Uparrow \lambda\alpha{:}\kappa.\mathcal{L}_1 \cup \mathcal{L}_2 \Rightarrow \tau' \restriction \mathcal{L}_3}{\mathcal{L};\{\!\{v_1 \bowtie v_2:\tau'\}\!\}^\pm_{l=\tau} \mapsto \mathcal{L};\{\!\{v_1:\lambda\alpha{:}\kappa.\mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L}_3\}\!\}^\pm_{l=\tau} \bowtie \{\!\{v_2:\lambda\alpha{:}\kappa.\mathcal{L}_2 \Rightarrow \tau' \restriction \mathcal{L}_3\}\!\}^\pm_{l=\tau}}$$

$$\textsf{ev:hcolor-else}\ \frac{\tau' \Uparrow \lambda\alpha{:}\kappa.\mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L}_2}{\begin{array}{l}\mathcal{L};\{\!\{\{\_ \Rightarrow e'\}:\tau'\}\!\}^\pm_{l=\tau} \mapsto \\ \quad \mathcal{L};\{\_ \Rightarrow \{\!\{e':\lambda\alpha{:}\kappa.\forall^+\chi.\forall\alpha{:}\chi \restriction \mathcal{U}.\tau'\langle\alpha:\chi \restriction \mathcal{U}\rangle\}\!\}^\pm_{l=\tau}\}\end{array}}$$

$$\textsf{ev:hcolor-tabs}\ \frac{\tau' \Uparrow \lambda\alpha{:}\kappa.\forall\beta{:}\kappa' \restriction \mathcal{L}'.\tau'}{\mathcal{L};\{\!\{\Lambda\beta{:}\kappa' \restriction \mathcal{L}'.e:\tau'\}\!\}^\pm_{l=\tau} \mapsto \mathcal{L};\Lambda\beta{:}\kappa' \restriction \mathcal{L}'.\{\!\{e:\lambda\alpha{:}\kappa.\tau'\}\!\}^\pm_{l=\tau}}$$

$$\textsf{ev:hcolor-kabs}\ \frac{\tau' \Uparrow \lambda\alpha{:}\kappa.\forall^+\chi.\tau'}{\begin{array}{l}\mathcal{L};\{\!\{\Lambda^+\chi.e:\tau'\}\!\}^\pm_{l=\tau} \mapsto \\ \quad \mathcal{L};\Lambda^+\chi.\{\!\{e:\lambda\alpha{:}\kappa.\tau'\}\!\}^\pm_{l=\tau}\end{array}} \qquad \textsf{ev:hcolor-sabs}\ \frac{\tau' \Uparrow \lambda\alpha{:}\kappa.\forall s{:}\mathrm{Ls}.\tau'}{\begin{array}{l}\mathcal{L};\{\!\{\Lambda s{:}\mathrm{Ls}.e:\tau'\}\!\}^\pm_{l=\tau} \mapsto \\ \quad \mathcal{L};\Lambda s{:}\mathrm{Ls}.\{\!\{e:\lambda\alpha{:}\kappa.\tau'\}\!\}^\pm_{l=\tau}\end{array}}$$

$$\textsf{ev:hcolor-labs}\ \frac{\tau' \Uparrow \lambda\alpha{:}\kappa.\forall^*\iota{:}\mathrm{L}(\kappa').\tau'}{\begin{array}{l}\mathcal{L};\{\!\{\Lambda^*\iota{:}\mathrm{L}(\kappa').e:\tau'\}\!\}^\pm_{l=\tau} \mapsto \\ \quad \mathcal{L};\Lambda^*\iota{:}\mathrm{L}(\kappa').\{\!\{e:\lambda\alpha{:}\kappa.\tau'\}\!\}^\pm_{l=\tau}\end{array}}$$

$$\textsf{ev:hcolor-color}\ \frac{\tau' \Uparrow \lambda\alpha{:}\kappa.\rho[l_1]}{\mathcal{L};\{\!\{\{\!\{v\}\!\}^+_{l_1=\tau_1}:\tau'\}\!\}^\pm_{l_2=\tau_2} \mapsto \mathcal{L};\{\!\{\{\!\{v:\lambda\alpha{:}\kappa.\rho[\tau_1]\}\!\}^\pm_{l_2=\tau_2}\}\!\}^+_{l_1=\tau_1}}$$

$$\textsf{ev:typecase1}\frac{\tau \Downarrow_* \rho[\mathtt{l}_i^\kappa] \qquad \{\mathtt{l}_i^\kappa \Rightarrow e'\} \in v \qquad \rho \rightsquigarrow p}{\mathcal{L}; \mathtt{typecase}\ \tau\ v \mapsto \mathcal{L}; p[e']}$$

$$\textsf{ev:typecase2}\frac{\tau \Downarrow_* \rho[\mathtt{l}_i^\kappa] \qquad \mathtt{l}_i^\kappa \notin v \qquad \{\_ \Rightarrow e'\} \in v}{\mathcal{L}; \mathtt{typecase}\ \tau\ v \mapsto \mathcal{L}; e'[\kappa]^+[\mathtt{l}_i^\kappa]}$$

$$\textsf{ev:setcase-empty}\frac{\mathcal{L} \downarrow_* \emptyset}{\mathcal{L}; \mathtt{setcase}\ \mathcal{L}\ e_\emptyset\ e_{\{\}}\ e_\cup\ e_\mathcal{U} \mapsto \mathcal{L}; e_\emptyset}$$

$$\textsf{ev:setcase-sing}\frac{\mathcal{L} \downarrow_* \{l\}}{\mathcal{L}; \mathtt{setcase}\ \mathcal{L}\ e_\emptyset\ e_{\{\}}\ e_\cup\ e_\mathcal{U} \mapsto \mathcal{L}; e_{\{\}}[\kappa]^+[l]^*}$$

$$\textsf{ev:setcase-union}\frac{\mathcal{L} \downarrow_* \mathcal{L}_1 \cup \mathcal{L}_2}{\mathcal{L}; \mathtt{setcase}\ \mathcal{L}\ e_\emptyset\ e_{\{\}}\ e_\cup\ e_\mathcal{U} \mapsto \mathcal{L}; e_\cup[\mathcal{L}_1][\mathcal{L}_2]}$$

$$\textsf{ev:setcase-univ}\frac{\mathcal{L} \downarrow_* \mathcal{U}}{\mathcal{L}; \mathtt{setcase}\ \mathcal{L}\ e_\emptyset\ e_{\{\}}\ e_\cup\ e_\mathcal{U} \mapsto \mathcal{L}; e_\mathcal{U}} \qquad \textsf{ev:lindex}\frac{}{\mathcal{L}; \mathtt{lindex}\ \mathtt{l}_i^\kappa \mapsto \mathcal{L}; i}$$

A.3.6 Congruence rules

$$\textsf{ev:app-con1}\frac{\mathcal{L}; e_1 \mapsto \mathcal{L}'; e_1'}{\mathcal{L}; e_1 e_2 \mapsto \mathcal{L}'; e_1' e_2} \qquad \textsf{ev:app-con2}\frac{\mathcal{L}; e \mapsto \mathcal{L}'; e'}{\mathcal{L}; v e \mapsto \mathcal{L}'; v e'} \qquad \textsf{ev:color-con}\frac{\mathcal{L}; e \mapsto \mathcal{L}'; e'}{\mathcal{L}; \{\!\{e\}\!\}^\pm_{l=\tau} \mapsto \mathcal{L}'; \{\!\{e'\}\!\}^\pm_{l=\tau}}$$

$$\textsf{ev:hcolor-con}\frac{\mathcal{L}; e \mapsto \mathcal{L}'; e'}{\mathcal{L}; \{\!\{e : \tau_1\}\!\}^\pm_{l=\tau_2} \mapsto \mathcal{L}'; \{\!\{e' : \tau_1\}\!\}^\pm_{l=\tau_2}} \qquad \textsf{ev:join-con1}\frac{\mathcal{L}; e_1 \mapsto \mathcal{L}'; e_1'}{\mathcal{L}; e_1 \bowtie e_2 \mapsto \mathcal{L}'; e_1' \bowtie e_2}$$

$$\textsf{ev:join-con2}\frac{\mathcal{L}; e \mapsto \mathcal{L}'; e'}{\mathcal{L}; v \bowtie e \mapsto \mathcal{L}'; v \bowtie e_2'} \qquad \textsf{ev:typecase-con}\frac{\mathcal{L}; e \mapsto \mathcal{L}'; e'}{\mathcal{L}; \mathtt{typecase}\ \tau\ e \mapsto \mathcal{L}'; \mathtt{typecase}\ \tau\ e'}$$