November 2004

An Open and Shut Typecase (Extended Version)

Dimitrios Vytiniotis
University of Pennsylvania

Geoffrey Washburn
University of Pennsylvania

Stephanie Weirich
University of Pennsylvania, sweirich@cis.upenn.edu

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-04-26.

An Open and Shut Typecase (Extended Version)

Abstract
Ad-hoc polymorphism is a compelling addition to typed programming languages. There are two different forms of ad-hoc polymorphism. With the nominal form, the execution of an operation is determined solely by the name of the type argument, whereas with the structural form, operations are defined by case analysis on the structure of types. The two forms differ in the way that they treat user-defined types. Operations defined by the nominal approach are considered "open"—the programmer can add cases for new types without modifying existing code. The operations must be extended however with specialized code for the new types, and it may be tedious and even difficult to add extensions that apply to a potentially large universe of user-defined types. Structurally defined operations apply to new types by treating them as equal to their underlying definitions, so no new cases for new types are necessary. However this form is considered "closed" to extension, as the behaviour of the operations cannot be differentiated for the new types. This form destroys the distinctions that user-defined types are designed to express. Both approaches have their benefits, so it is important to provide both capabilities in a single language that is expressive enough to decouple the "openness" issue from the way that user-defined types are treated. We present such a language that supports both forms of ad-hoc polymorphism.

Comments
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-04-26.

An Open and Shut Typecase (Extended Version)

Dimitrios Vytiniotis Geoffrey Washburn Stephanie Weirich
Technical Report MS-CIS-04-26
Department of Computer and Information Science
University of Pennsylvania
{dimitriv, geoffw, sweirich}@cis.upenn.edu

November 4, 2004

Abstract

Ad-hoc polymorphism is a compelling addition to typed programming languages. There are two different forms of ad-hoc polymorphism. With the nominal form, the execution of an operation is determined solely by the name of the type argument, whereas with the structural form, operations are defined by case analysis on the structure of types. The two forms differ in the way that they treat user-defined types. Operations defined by the nominal approach are considered "open"—the programmer can add cases for new types without modifying existing code. The operations must be extended however with specialized code for the new types, and it may be tedious and even difficult to add extensions that apply to a potentially large universe of user-defined types. Structurally defined operations apply to new types by treating them as equal to their underlying definitions, so no new cases for new types are necessary. However this form is considered "closed" to extension, as the behaviour of the operations cannot be differentiated for the new types. This form destroys the distinctions that user-defined types are designed to express. Both approaches have their benefits, so it is important to provide both capabilities in a single language that is expressive enough to decouple the "openness" issue from the way that user-defined types are treated. We present such a language that supports both forms of ad-hoc polymorphism.

1 Introduction

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

Ad-hoc polymorphism is a compelling addition to a typed programming language. 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. Therefore, ad-hoc polymorphism can significantly simplify programming with complicated data structures, eliminating the need for repetitive "boilerplate code".

Currently, there are two forms of ad-hoc polymorphism in typed, functional languages. The first is based on the *nominal* analysis of type information, such as Haskell type classes [30]. In this approach, the execution of an ad-hoc operation is determined solely by the names of the head constructors of the type arguments.

Consider for example the implementation of an equality function using Haskell type classes. The type class declares that there is a type-directed operation called `eq`.

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

Each instance of the class describes how `eq` behaves for that type. For composite types, such as products or lists, equality is defined in terms of equality for the components of the type.

```
instance Eq Int where
  eq x y = eqint x y
instance Eq Bool where
  eq x y = if x then y else not y
instance (Eq a, Eq b) => Eq (a,b) where
  eq (x1,y1) (x2,y2) = eq x1 y1 && eq x2 y2
instance (Eq a) => Eq [a] where
  eq l1 l2 = all2 eq l1 l2
```

Note that nominal analysis naturally limits the domain of ad-hoc operations to those types where a definition has been provided. For example, `eq` is not defined for function types. Type-directed operations in such a framework are considered "open"; at any time the programmer may extend them with instances for new types, without modifying existing code.

The second form of ad-hoc polymorphism is based on *structural* analysis of type information. In intensional type analysis [11] programmers define type-directed operations by case analysis on the type structure. Polymorphic equality defined in this approach may look like the following:

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

Because type-directed operations are defined by case analysis, they are considered "closed" to extension, that is, the programmer does not have a way to modify or extend the behaviour of polymorphic operations to new types. Moreover, cases for all types must be provided when such operations are defined; even when some cases are nonsensical for the particular operation.

The two forms of ad-hoc polymorphism differ in the way that they treat user-defined types. User-defined types such as Haskell's `newtypes` [23], are an important feature of many languages. Although these new types are isomorphic to existing types, they express application-specific distinctions that can be enforced 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 have the same underlying integer representation.

In the nominal approach, the type-directed operations must be extended with instances for each new user-defined type, because the new types are not equivalent to their underlying structural definition. Moreover, if some types are defined in separate inaccessible modules, then it is impossible for the programmer to extend the operation to those types. Instead, he must rely on the definer of the type to add the instance; but there is no guarantee that the definer of the type will indeed respect the invariants of the type-directed operation.

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

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

1.1 Combining both forms in one language

We attempt to unify 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. It allows for the developers to choose which characteristics they want to use from each system, instead of forcing them to make this decision ahead of time. This language allows the "openness to extension" from the way that user-defined types are treated. We show that the notions of "nominal versus structural" and "open versus closed" become orthogonal as soon as the underlying calculus becomes expressive enough.

At the core, $\lambda_{\mathcal{L}}$ is a system for structural type analysis augmented with user-defined types. The structural type analysis operator **typecase** may include branches for new names, representing new user-defined types. Types containing names for which there is no branch in an operation cannot be allowed as arguments, 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.

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

Additionally, $\lambda_{\mathcal{L}}$ includes support for coercion of the types of expressions so that new type names that are mentioned in these types are replaced by their underlying definitions.

The goal of $\lambda_{\mathcal{L}}$ is to provide a unifying calculus or a typed intermediate representation for languages that support type-directed operations.

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

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

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

1.2 Contributions of this work

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

- We define a language that allows the definition of both "open" and "closed" type-directed operations. Previous work has chosen one or the other, augmented with ad-hoc mechanisms to counter their difficulties.
- We define a language that allows programmers to statically restrict the domain of type-directed operations defined in a structural system in a natural manner. Previous work [11, 4] requires that programmers use type-level analysis or programming to makes such restrictions.
- We show how to reconcile **typecase** with the analysis of higher-order type constructors. Previous work [32] has based such analysis on the interpretation of type constructors. In $\lambda_{\mathcal{L}}$, we show how to implement the same operations with simpler constructs.
- We present a sophisticated system of coercions for converting between new types and their definitions. We extend previous work [23, 29, 26] to higher-order coercions.

The remainder of this technical report is as follows. In the next section we introduce the features of $\lambda_{\mathcal{L}}$ through examples. We first describe the semantics of the core language in Section 3, and then extend it to be fully reflexive in Section 4. We discuss additional extensions in Section 5. We summarize some related work

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

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

in Section 6 and conclude in Section 7. We give the semantics of the fully reflexive language in Appendix A and we give proofs for the properties of the core $\lambda_{\mathcal{L}}$ in Appendix B.

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

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

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

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

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

2.1 Generative types

The $\lambda_{\mathcal{L}}$ language includes a simple mechanism that allows users to define new type constants. We call all type constants *labels* to emphasize the fact that they do not α-vary. Arbitrary label constants are taken from an enumerable set and written as ℓ_i^κ, parameterized by their kind κ and their index i. Some distinguished constants in this language are constructors for primitive types. The label $\ell_0^\star$ is a nullary constructor for the type of integers, and $\ell_1^{\star\to\star\to\star}$ is the the binary constructor for function types. We use the syntactic sugar ℓ_{int} and $\ell_\to$ to refer to these two labels. However, when these labels appear in types, we use the notation int to stand for ℓ_{int} and $\tau_1 \to \tau_2$ to stand for the function type $\ell_\to\ \tau_1\ \tau_2$. In the examples, we extend this language with new forms of types, such as booleans (bool), products ($\tau_1 \times \tau_2$), and lists ($\mathsf{list}\ \tau$), and add new label constants, written ℓ_{bool}, $\ell_\times$ and ℓ_{list}, to form these types.

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

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

Unlike other forms of user-defined types, such as Haskell `newtype`s, this mechanism dynamically creates new "types". Generating these new labels requires an operational effect at run time. However, the coercions that convert between the new label and its definition have no run-time cost.

Note that even though run-time type analysis destroys the parametricity induced by type abstractions, users may still hide the implementation details of abstract datatypes using generative types. Once outside the scope of a new label, it is impossible to determine its underlying definition. For example, we know that the polymorphic function f below must treat its term argument parametrically because, even in the presence of run-time type analysis, it cannot coerce it to the type int.

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

2.2 Type analysis with a restricted domain

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

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

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

For this reason, when type checking a **typecase** expression, $\lambda_{\mathcal{L}}$ calculates the set of labels that may appear within the analyzed type and requires that set to be a subset of the set of labels that apear in the domains of maps in **typecase**. Label sets in $\lambda_{\mathcal{L}}$ may be empty, $\varnothing$, may contain a single label, $\{l\}$, may be the union of two label sets, $\mathcal{L}_1 \cup \mathcal{L}_2$, or may be the entire universe of labels, $\mathcal{U}$. Analogously to finite maps, $\{l_1, \ldots, l_n\}$ abbreviates $\{l_1\} \cup \ldots \cup \{l_n\}$.

To allow type polymorphism, we annotate a quantified type variable with the set of labels that may appear in types that instantiate it. For example, below we know that α will be instantiated only by a type

formed from the labels ℓ_{int} and ℓ_{bool}(i.e., by int or bool), so α will have a match in the **typecase** expression.

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

If we annotate a type variable with $\mathcal{U}$ then it is unanalyzable because no **typecase** can cover all branches.[1]

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

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

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

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

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

2.3 Generative types and type analysis

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

The reconciliation of type analysis with the dynamic creation of new type names is the fundamental problem addressed by $\lambda_{\mathcal{L}}$.

How can we apply a function like **eq** to types that mention newly generated labels? There are two scenarios. The programmer may wish to implement a polytypic function so that:

- it is not applicable to any argument that uses a type name for which it does not have branch. Such terms must first be coerced to a type-isomorphic version mentioning only supported type names before it may be passed as an argument.
  or

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

- it is extensible with new branches for the new type names. Even though new types names may be isomorphic to existing types, there is a programmer-defined distinction between values of the new type and values of the underlying representation, and polytypic operations must treat these new types in a special manner. For example, even though the type telephone may be isomorphic to the type int, a polytypic pretty-printer should display telephone numbers differently from integers.

  We have already discussed the first solution that uses coercions in subsection 2.1 and we are going to see a natural extension of this in subsection 2.3.2. The second solution is discussed in subsection 2.3.1.

2.3.1 Extensible type analysis

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

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

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

Because existing maps may be shadowed in joins, type directed operations can be possibly redefined for those names that belong in the domain of the maps that get shadowed. Redefining the behavior of **typecase** for int may not be what the programmer intended, but allowing such a scenario does not affect the soundness of $\lambda_{\mathcal{L}}$. If the programmer wished to prevent this redefinition, she could join x on the left.

However, even if a type-directed function abstracts a map for **typecase**, it is still not extensible. The type of that map specifies the labels that are in its domain. Branches for newly created labels cannot be supplied. This restriction does not complement our language of dynamic label creation very well—we may wish to apply a polytypic function to types that contain labels defined outside the scope of the function, by supplying a map for these labels.

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

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

An extension for products would certainly have to call the extended version of **eq** for the components of the products, hence it would have to be recursive.

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

We call the extended equality function as follows.

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

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

There are two ways to write **tostring**. The first is to supply the branches for **important** as an additional argument to **tostring**, as below.

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

Dependency-Style Generic Haskell [20] uses this technique. In that language, the additional arguments are automatically inferred by the compiler. However, the dependencies still show up in the type of an operation, hindering the modularity of the program.

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

2.3.2 Higher-order coercions

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

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

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

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

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

Higher-order coercions have no run-time effect; they merely alter the types of expressions, by replacing between labels and their definitions in argument positions in type applications. For reasons of type checking, a higher-order coercion is annotated with a type constructor—in this case list—that describes the location of the label to coerce in the type of the term.

2.4 Higher-order type analysis

Higher-order type analysis [32] is often used to define operations in terms of parameterized data structures, such as *lists* and *trees*. These operations must be able to distinguish between the type parameter and the rest of the type.

We present a characteristic example, due to Hinze [12]. Let $\mathcal{L} = \{\ell_{\mathsf{int}}, \ell_{\mathsf{bool}}, \ell_{\times}\}$ and consider the following open function:

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

The function is seemingly useless, but suppose that we have a type constructor τ, of kind $\star \to \star$:

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

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

$$
\begin{array}{l}
\textbf{delegate} = \Lambda\gamma{:}\star \to \star \restriction \mathcal{L}.\ \Lambda\alpha{:}\star \restriction \mathcal{L}. \\
\quad \lambda f{:}\alpha \to \mathsf{int}. \\
\quad\quad \textbf{new}\ \iota{:}\star = \alpha\ \textbf{in} \\
\quad\quad\quad \lambda x : \gamma\ \alpha.\ \textbf{ecount}[\{\iota\}] \\
\quad\quad\quad\quad\quad \{\iota \quad \Rightarrow \quad \lambda x{:}\iota.\ f\ \{\!\{x\}\!\}^{+}_{\iota}\}\ \ [\gamma\ \iota]\ \{\!\{x : \gamma\}\!\}^{-}_{\iota}
\end{array}
$$

The function abstracts the constructor γ, its type parameter α, and then takes a function f describing how the operation should behave for α. It then creates a new label ι, isomorphic to α, and takes an argument of type $\gamma\ \alpha$. The argument type is coerced to $\gamma\ \iota$ and, finally, the function **ecount** is applied to the argument, taking an extension for label ι that calls f. Now, here's how we can effectively compute the number of data nodes of our data structure. The following call will return 2.

$$\textbf{delegate}\ [\tau][\mathsf{int}](\lambda x{:}\mathsf{int}.\ 1)((1,2),2)$$

The style of open definitions can be used to encode other useful examples that require higher-order analysis, such as an extensible type-safe cast operator, based on Weirich's functional pearl [31]. The reader is invited to study the examples under the `examples/` directory of the implementation.

3 The Core Language

Next we describe the semantics of core $\lambda_{\mathcal{L}}$ in detail. The complete semantics of a fully reflexive variant of $\lambda_{\mathcal{L}}$ appears in Appendix A, while the semantics of core $\lambda_{\mathcal{L}}$ appears in Appendix B. In Figure 2 we present some extra syntactic categories necessary for the presentation of the dynamic and static semantics. Type and term contexts are as expected. Type isomorphisms Σ are used to record the isomorphisms between labels and types and are introduced by **new** expressions. Type paths ρ are simply type-level applications of a hole to a sequence of types, and term paths p are term-level applications of a hole to a sequence of types.

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

Figure 2: Extra syntactic categories

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

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

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

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

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

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

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

The judgment $\mathcal{L}; e \mapsto \mathcal{L}'; e'$ describes the small-step call-by-value operational semantics of the language. A term e with a set of labels $\mathcal{L}$ steps to a new term e' possibly with larger set of labels $\mathcal{L}'$. During the evaluation of the **new** operator, a fresh label constant is generated and added to the label set component. In this way, it resembles an *allocation semantics* [22, 8]. The initial state of execution includes all label constants, such as ℓ_{int} and $\ell_{\to}$, 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.

3.1 Semantics of generative types

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

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

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

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

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

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

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

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

Higher-order coercions allow the non-head positions of a type to change. These coercions are annotated with a type constructor τ' that describes the shape of the data structure to be coerced.

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

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

$$\frac{\Delta \vdash \tau : \star \qquad \Delta \vdash \tau \Downarrow_{*} \tau' \qquad \Delta \vdash \tau' \not\Downarrow}{\Delta \vdash \tau \downarrow \tau'} \qquad \frac{\Delta \vdash \tau : \kappa_1 \to \kappa_2 \qquad \Delta, \alpha{:}\kappa_1 \vdash \tau\ \alpha \downarrow \tau'}{\Delta \vdash \tau \downarrow \lambda\alpha{:}\kappa_1.\tau}$$

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

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

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

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

3.2 Semantics of type analysis

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

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

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

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

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

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

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

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

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

Once the rule for type checking **typecase** determines the labels that could appear in the argument type, it looks at the type of the first-class map to determine the domain of the map. Given some map e with domain $\mathcal{L}_1$ and a type argument τ that mentions labels in $\mathcal{L}$, this rule checks that the map can handle all possible labels in τ with $\mathcal{L} \sqsubseteq \mathcal{L}_1$.

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

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

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

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

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

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

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

3.3 Properties

The $\lambda_{\mathcal{L}}$ language is type sound, following from the usual progress and preservation theorems [33]. The proofs of these theorems are inductions over the relations defined.

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

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

We have also shown that the coercions are not necessary to the operational semantics. A calculus where the coercions have been erased has the same operational behaviour as this calculus. In other words, expressions in $\lambda_{\mathcal{L}}$ evaluate to a value if and only if their coercion-erased versions evaluate to the coercion-erased value. This justifies our claim that coercions have no operational effect—even though the naive evaluation rules for coercions presented in this section do. The proofs of the above statements can be found in Appendix B.

4 Full Reflexivity

The core language does not offer the capability of full reflexivity. Some types cannot be analyzed by **typecase**. The full $\lambda_{\mathcal{L}}$ language addresses this problem and extends the set of analyzable types to include all types. It also includes label and label set runtime analysis operators. In the rest of this section we discuss these extensions. The modifications to the syntax of core $\lambda_{\mathcal{L}}$ to support full reflexivity appear in Figure 4.

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

There is one implication in the addition of these new abstraction forms. Polykinded types cannot be determined statically in general, as the kind over which they are parameterized may be unknown at compile time. Therefore polykinded types are part of the syntax of the full language, instead of being derived forms. A type equivalence relation encodes the fact that they are equivalent to certain simpler types. The interesting equivalences are given in Figure 6. Notice that polykinded types do not have a label constructor in Figure 7. At run time, closed polykinded types will always be reduced to one of the other type forms.

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

Figure 4: Modifications for full reflexivity

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

Figure 5: Distinguished label kinds

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

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

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

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

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

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

Figure 6: Polykinded type equivalences

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

Figure 7: Syntactic sugar for types

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

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

In this rule, $e_{\{\}}$ must be able to take any label as its argument, whatever the kind of the label. Therefore $\lambda_{\mathcal{L}}$ must support kind polymorphism, as shown in Figure 4.

5 Extensions

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

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

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

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

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

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

$$\begin{array}{l}[\![\{\!\{\mathbf{inr}\,\langle 1, \{\!\{\mathbf{inr}\,\langle 3, \{\!\{\mathbf{inl}\,\langle\rangle\}\!\}^{+}_{\iota}\rangle\}\!\}^{+}_{\iota}\rangle\}\!\}^{+}_{\iota} : \lambda\alpha{:}\star.\alpha]\!]^{-}_{\iota=1+\mathsf{int}\times\iota} \mapsto^{*} \\ \quad [1 + \mathsf{int} \times (1 + \mathsf{int} \times (1 + \mathsf{int})), \mathbf{inr}\,\langle 1, \mathbf{inr}\,\langle 3, \mathbf{inl}\,\langle\rangle\rangle\rangle] \ \mathbf{as} \\ \quad\quad \exists\alpha{:}\star \restriction \{\ell_{\times}, \ell_{+}, \ell_1, \ell_{\mathsf{int}}\}.\alpha\end{array}$$

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

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

The extension that we need for record and variant types is *finite type maps* from from labels to types of kind $\star$. Finite type maps are new syntactic category with their own form of abstraction and application in both the type and term languages, as well as finite map analysis. Rules analogous to those for label set subsumption, membership and equality can be defined for these finite maps. The distinguished label ℓ_{rec} of kind (Map $\to \star$) forms record types from finite maps. As with many versions of records, these types are equivalent up to permutation. Record terms are formed from empty records $\varnothing$, singletons $\{l = e\}$, or concatenation $e_1 \circ e_2$. If l is in the domain of the record type, the record projection $e.l$ is well-formed. Because we provide abstractions over finite maps, these records get a form of row polymorphism [24] for free. It is straightforward to develop similar extensions for variants.

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

6 Related work

There is much research on type-directed programming. *Run-time type analysis* allows the structural analysis of dynamic type information. Abadi, et al. introduced a type-dynamic to which types could be coerced, and later via case analysis, extracted [1]. The core semantics of **typecase** in $\lambda_{\mathcal{L}}$ is similar to the intensional polymorphism of Harper and Morrisett [11]. However, $\lambda_{\mathcal{L}}$ does not include a type-level analysis operator. Our extension of $\lambda_{\mathcal{L}}$ to be fully reflexive follows a similar extension of Harper and Morrisett's language by Trifonov, Saha, and Shao [28]. Weirich [32] extended run-time analysis to higher-order type constructors following the work of Hinze [12].

Generic programming uses the structure of datatypes to generate specialized operations at compile time. The Charity language [3] automatically generates folds for datatypes. PolyP [15] is an extension of Haskell that allows the definition of polytypic operations based on positive, regular datatypes. Functorial ML [17] bases polytypic operations on the composition of functors, and has lead to the programming language FISh [16]. Generic Haskell [2], following the work of Hinze [12] allows polytypic functions to be indexed by any type or type constructor.

Nominal forms of ad-hoc polymorphism are usually used for *overloading*. Type classes in Haskell [30] implement overloading by defining classes of types that have instances for a set of polytypic operations. Hinze and Peyton Jones [13] explored an extension to automatically derive type class instances by looking at the underlying structure of new types. Dependency-style Generic Haskell [20] revises the Generic Haskell

language to be based on the names of types instead of their structure. However, to automatically define more generic functions, it converts user-defined types into their underlying structural representations if a specific definition has not been provided.

Many languages use a form of generative types to represent application-specific abstractions. For example, Standard ML [21] and Haskell [23] rely on datatype generativity in type inference. Modern module systems also provide generative types [5]. When the definition of the new type is known, the type isomorphisms we present differ from calculi with type equalities (such as provided by Harper and Lillibridge [10] or Stone and Harper [27]) in that they require explicit terms to coerce between a type name and its definition. While explicit coercions are more difficult for the programmer to use, they simplify the semantics of the generative types.

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

7 Discussion

In conclusion, the $\lambda_{\mathcal{L}}$ language provides a framework that can model 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.

There are certain drawbacks in $\lambda_{\mathcal{L}}$. Although the flexibility of having unanalyzable types—by using $\mathcal{U}$ in type abstractions—is important, this is not the best way to it does not allow types to be partly abstract and partly transparent. Even when a type is treated parametrically most of the time, it still has to be annotated with a label set. To overcome this, we have to come up with a label set inference mechanism, similar to Haskell's inference mechanism for type classes.

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

In order to increase the expressiveness of the core language, we plan to extend it in two ways. First, **typecase** makes restrictions on all labels that appear in its argument so that it can express catamorphisms over the structure of the type language. However, not every type-directed function is a catamorphism. Some operations only determine the head form of the type. Others are hybrids, applicable to a specific pattern of type structure. For example, if we were to add references to the calculus, we could extend **eq** to all references, even if their contents are not comparable, by using pointer equality. This calculus cannot express that pattern. Furthermore, some operations are only applicable to very specific patterns. For example, an operation may be applicable only to functions that take integers as arguments, such as functions of the form $\mathsf{int} \to \mathsf{int}$ or $\mathsf{int} \to \mathsf{int} \to \mathsf{int}$. These operations are still expressible in the core calculus, but there is no way to statically determine whether the type argument satisfies one of these patterns, so dynamic checks must be used. To approach this problem, we plan to investigate pattern calculi that may be able to more precisely specify the domain of type-directed operations. For example, the mechanisms of languages designed to support native XML processing [14, 6] can statically enforce that tree-structured data has a very particular form.

Finally, it is also important to add type-level analysis of types. As shown in past work, it is impossible

to assign types to some type-directed functions without this feature. One way to do so might be to extend the primitive-recursive operator of Trifonov et al. [28] to include first-class maps from labels to types.

Acknowledgments

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[31] S. Weirich. Type-safe cast: Functional pearl. In *Proceedings of theFifth International Conference on Functional Programming (ICFP)*, pages 58–67, Montreal, Sept. 2000.

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

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

$Id: tech-report.tex 364 2004-11-04 17:45:58Z dimitriv $

A Full Language

A.1 Syntax

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

A.2 Judgments

Static Judgments	
Kind well-formedness	$\Delta \vdash \kappa$
Label well-formedness	$\Delta \vdash l : \mathrm{L}(\kappa)$
Label set well-formedness	$\Delta \vdash \mathcal{L} : \mathrm{Ls}$
Label set subsumption	$\Delta \vdash \mathcal{L}_1 \sqsubseteq \mathcal{L}_2$
Label set equivalence	$\Delta \vdash \mathcal{L}_1 = \mathcal{L}_2$
Type well-formedness	$\Delta \vdash \tau : \kappa$
Type label set analysis	$\Delta \vdash \tau \,|\, \mathcal{L}$
Type equivalence	$\Delta \vdash \tau_1 = \tau_2 : \kappa$
Term well-formedness	$\Delta; \Gamma \vdash e : \tau \,|\, \Sigma$

Dynamic Judgments	
Small-step evaluation	$\mathcal{L}; e \mapsto \mathcal{L}'; e'$
Weak-head reduction	$\Delta \vdash \tau \Downarrow \tau'$
Weak-head normalization	$\Delta \vdash \tau \downarrow \tau'$
Label set reduction	$\mathcal{L}_1 \downarrow \mathcal{L}_2$
Path conversion	$\rho \rightsquigarrow p$

A.3 Static semantics

A.3.1 Kind well-formedness

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

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

A.3.2 Label well-formedness

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

A.3.3 Label set well-formedness

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

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

A.3.4 Type well-formedness

$$\frac{\alpha{:}\kappa \in \Delta}{\Delta \vdash \alpha : \kappa}\ \textsf{twf:var} \qquad \frac{\alpha{:}\kappa \restriction \mathcal{L} \in \Delta}{\Delta \vdash \alpha : \kappa}\ \textsf{twf:var-res} \qquad \frac{\Delta \vdash \tau_1 : \kappa_1 \to \kappa_2 \qquad \Delta \vdash \tau_2 : \kappa_1}{\Delta \vdash \tau_1\tau_2 : \kappa_2}\ \textsf{twf:app}$$

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

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

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

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

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

A.3.5 Type label set analysis

$$\frac{\alpha{:}\kappa \in \Delta}{\Delta \vdash \alpha \mid \varnothing}\ \textsf{tan:var} \qquad \frac{\alpha{:}\kappa \restriction \mathcal{L} \in \Delta}{\Delta \vdash \alpha \mid \mathcal{L}}\ \textsf{tan:var-res} \qquad \frac{\Delta \vdash \tau_1 \mid \mathcal{L}_1 \qquad \Delta \vdash \tau_2 \mid \mathcal{L}_2}{\Delta \vdash \tau_1\tau_2 \mid \mathcal{L}_1 \cup \mathcal{L}_2}\ \textsf{tan:app}$$

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

$$\frac{\Delta \vdash \tau \mid \mathcal{L}_1 \qquad \Delta \vdash \tau' \mid \mathcal{L}_2}{\Delta \vdash \tau'\langle \tau : \kappa \restriction \mathcal{L}\rangle \mid \mathcal{L}_1 \cup \mathcal{L}_2}\ \textsf{tan:polyk} \qquad \frac{}{\Delta \vdash l \mid \{l\}}\ \textsf{tan:ltype}$$

$$\frac{\Delta, \chi \vdash \tau \mid \mathcal{L}}{\Delta \vdash \Lambda\chi.\tau \mid \mathcal{L}}\ \textsf{tan:kabs} \qquad \frac{\Delta \vdash \tau \mid \mathcal{L}}{\Delta \vdash \tau[\kappa_1] \mid \mathcal{L}}\ \textsf{tan:kapp} \qquad \frac{\Delta, \iota{:}\mathrm{L}(\kappa_1) \vdash \tau \mid \mathcal{L}}{\Delta \vdash \lambda\iota{:}\mathrm{L}(\kappa_1).\tau \mid \mathcal{L}}\ \textsf{tan:labs}$$

$$\frac{\Delta \vdash \tau \mid \mathcal{L}}{\Delta \vdash \tau\hat{l} \mid \mathcal{L}}\ \textsf{tan:lapp} \qquad \frac{\Delta, s{:}\textsc{Ls} \vdash \tau \mid \mathcal{L}}{\Delta \vdash \lambda s{:}\textsc{Ls}.\tau \mid \mathcal{L}}\ \textsf{tan:sabs} \qquad \frac{\Delta \vdash \tau \mid \mathcal{L}_1}{\Delta \vdash \tau\mathcal{L}_2 \mid \mathcal{L}_1}\ \textsf{tan:sapp}$$

A.3.6 Label set subsumption

$$\frac{\Delta \vdash \mathcal{L} : \mathrm{Ls}}{\Delta \vdash \mathcal{L} \sqsubseteq \mathcal{L}}\ \textsf{ss:refl} \qquad \frac{\Delta \vdash \mathcal{L}_1 \sqsubseteq \mathcal{L}_2 \qquad \Delta \vdash \mathcal{L}_2 \sqsubseteq \mathcal{L}_3}{\Delta \vdash \mathcal{L}_1 \sqsubseteq \mathcal{L}_3}\ \textsf{ss:trans}$$

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

$$\frac{\Delta \vdash \mathcal{L} \sqsubseteq \mathcal{L}_1 \qquad \Delta \vdash \mathcal{L}_2 : \mathrm{Ls}}{\Delta \vdash \mathcal{L} \sqsubseteq \mathcal{L}_1 \cup \mathcal{L}_2}\ \textsf{ss:union-right1} \qquad \frac{\Delta \vdash \mathcal{L} \sqsubseteq \mathcal{L}_2 \qquad \Delta \vdash \mathcal{L}_1 : \mathrm{Ls}}{\Delta \vdash \mathcal{L} \sqsubseteq \mathcal{L}_1 \cup \mathcal{L}_2}\ \textsf{ss:union-right2}$$

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

A.3.7 Label set equivalence

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

A.3.8 Type equivalence

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A.3.9 Term well-formedness

$$\frac{x{:}\tau \in \Gamma}{\Delta; \Gamma \vdash x : \tau \mid \Sigma}\ \textsf{wf:var} \qquad \frac{}{\Delta; \Gamma \vdash i : \ell_{\mathsf{int}} \mid \Sigma}\ \textsf{wf:int} \qquad \frac{\Delta \vdash \tau_1 : \star \qquad \Delta; \Gamma, x{:}\tau_1 \vdash e : \tau_2 \mid \Sigma}{\Delta; \Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2 \mid \Sigma}\ \textsf{wf:abs}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \mid \Sigma \qquad \Delta; \Gamma \vdash e_2 : \tau_1 \mid \Sigma}{\Delta; \Gamma \vdash e_1 e_2 : \tau_2 \mid \Sigma}\ \textsf{wf:app} \qquad \frac{\Delta \vdash \tau : \star \qquad \Delta; \Gamma, x{:}\tau \vdash e : \tau \mid \Sigma}{\Delta; \Gamma \vdash \mathbf{fix}\ x{:}\tau.e : \tau \mid \Sigma}\ \textsf{wf:fix}$$

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

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

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

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

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

$$\frac{\Delta \vdash \mathcal{L} : \mathrm{Ls} \qquad \Delta \vdash l : \mathrm{L}(\kappa) \qquad \Delta; \Gamma \vdash e : \tau' \langle l : \kappa \restriction \mathcal{L} \rangle \mid \Sigma}{\Delta; \Gamma \vdash \{l \Rightarrow e\} : \{l\} \Rightarrow \tau' \restriction \mathcal{L} \mid \Sigma} \text{ wf:env-branch}$$

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

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

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

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

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

$$\frac{\Delta; \Gamma \vdash e : \forall \alpha{:}\kappa \restriction \mathcal{L}. \tau_1 \mid \Sigma \qquad \Delta \vdash \tau_2 : \kappa \qquad \Delta \vdash \tau_2 \mid \mathcal{L}' \qquad \Delta \vdash \mathcal{L}' \sqsubseteq \mathcal{L}}{\Delta; \Gamma \vdash e[\tau_2] : \tau_1[\tau_2/\alpha] \mid \Sigma} \text{ wf:tapp}$$

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

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

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

A.4 Dynamic semantics

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

A.4.1 Weak-head reduction for types

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

$$\frac{\Delta \vdash \tau_1 \Downarrow \tau_1'}{\Delta \vdash \tau_1\tau_2 \Downarrow \tau_1'\tau_2}\ \textsf{whr:app-con} \qquad \frac{}{\Delta \vdash (\lambda\iota{:}\mathrm{L}(\kappa).\tau)\hat{l} \Downarrow \tau[l/\iota]}\ \textsf{whr:labs-beta} \qquad \frac{\Delta \vdash \tau \Downarrow \tau'}{\Delta \vdash \tau\hat{l} \Downarrow \tau'\hat{l}}\ \textsf{whr:lapp-con}$$

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

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

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

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

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

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

A.4.2 Weak-head normalization for types

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

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

A.4.3 Reduction for label sets

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

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

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

A.4.4 Label set normal forms

$$\frac{}{\varnothing\ norm}\ \textsf{ln:empty} \qquad \frac{}{\mathcal{U}\ norm}\ \textsf{ln:univ} \qquad \frac{}{\{\ell_i^\kappa\}\ norm}\ \textsf{ln:sing}$$

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

A.4.5 Path conversion

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

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

A.4.6 Computation rules

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\mathcal{L} \downarrow_* \mathcal{U} \qquad \theta = \{\varnothing \Rightarrow e_\varnothing,\ \{\} \Rightarrow e_{\{\}}, \cup \Rightarrow e_\cup,\ \mathcal{U} \Rightarrow e_\mathcal{U}\ \}}{\mathcal{L}; \mathbf{setcase}\ \mathcal{L}\ \theta \mapsto \mathcal{L}; e_\mathcal{U}}\ \textsf{ev:setcase-top} \qquad \frac{}{\mathcal{L}; \mathbf{lindex}\ \ell_i^\kappa \mapsto \mathcal{L}; i}\ \textsf{ev:lindex}$$

A.4.7 Congruence rules

$$\frac{\mathcal{L}; e_1 \mapsto \mathcal{L}'; e_1'}{\mathcal{L}; e_1 e_2 \mapsto \mathcal{L}'; e_1' e_2}\ \textsf{ev:app-con1} \qquad \frac{\mathcal{L}; e \mapsto \mathcal{L}'; e'}{\mathcal{L}; v e \mapsto \mathcal{L}'; v e'}\ \textsf{ev:app-con2}$$

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

$$\frac{\mathcal{L}; e_1 \mapsto \mathcal{L}'; e_1'}{\mathcal{L}; e_1 \bowtie e_2 \mapsto \mathcal{L}'; e_1' \bowtie e_2}\ \textsf{ev:join-con1} \qquad \frac{\mathcal{L}; e_2 \mapsto \mathcal{L}'; e_2'}{\mathcal{L}; v \bowtie e \mapsto \mathcal{L}'; v \bowtie e_2'}\ \textsf{ev:join-con2}$$

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

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

B Core Language

B.1 Syntax

Kinds	κ	$::= \star$	*normal types*
		$\mid \kappa_1 \to \kappa_2$	*function kinds*
Labels	l	$::= \ell_i^\kappa$	*constants*
		$\mid \iota$	*variables*
label sets	$\mathcal{L}$	$::= \varnothing$	*empty*
		$\mid \{l\}$	*singleton*
		$\mid s$	*variable*
		$\mid \mathcal{L}_1 \cup \mathcal{L}_2$	*join*
		$\mid \mathcal{U}$	*universe*
Types	σ, τ	$::= \alpha \mid \lambda\alpha{:}\kappa.\tau \mid \tau_1\tau_2$	*λ-calculus*
		$\mid \forall\iota{:}\mathrm{L}(\kappa).\tau$	*label quantification*
		$\mid \forall\alpha{:}\kappa \restriction \mathcal{L}.\tau$	*type quantification*
		$\mid \forall s{:}\mathrm{Ls}.\tau$	*label set quantification*
		$\mid l$	*label coercion*
		$\mid \mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L}_2$	*branch types*
Terms	e	$::= x \mid \lambda x{:}\sigma.e \mid e_1 e_2$	*λ-calculus*
		$\mid \mathbf{fix}\ x{:}\sigma.e$	*recursion*
		$\mid i$	*integers*
		$\mid \mathbf{new}\ \iota{:}\kappa = \tau\ \mathbf{in}\ e$	*dynamic label creation*
		$\mid \{\!\{e\}\!\}^{\pm}_{l=\tau}$	*primitive coercions*
		$\mid \{\!\{e : \tau\}\!\}^{\pm}_{l=\tau_2}$	*extended coercions*
		$\mid \mathbf{typecase}\ \tau\ e$	*type case analysis*
		$\mid \mathbf{lindex}\ l$	*label analysis*
		$\mid \Lambda\alpha{:}\kappa \restriction \mathcal{L}.e \mid e[\tau]$	*type polymorphism*
		$\mid \Lambda\iota{:}\mathrm{L}(\kappa).e \mid e[l]$	*label polymorphism*
		$\mid \Lambda s{:}\mathrm{Ls}.e \mid e[\mathcal{L}]$	*label set polymorphism*
		$\mid \varnothing \mid \{l \Rightarrow e\} \mid e_1 \bowtie e_2$	*branches*

B.2 Polykinded types as derived forms

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

B.3 Judgments

Static Judgments	
Label well-formedness	$\Delta \vdash l : \mathrm{L}(\kappa)$
Label set well-formedness	$\Delta \vdash \mathcal{L} : \mathrm{Ls}$
Label set subsumption	$\Delta \vdash \mathcal{L}_1 \sqsubseteq \mathcal{L}_2$
Type well-formedness	$\Delta \vdash \tau : \kappa$
Type label set analysis	$\Delta \vdash \tau \,|\, \mathcal{L}$
Type equivalence	$\Delta \vdash \tau_1 = \tau_2 : \kappa$
Term well-formedness	$\Delta; \Gamma \vdash e : \sigma \,|\, \Sigma$

Dynamic Judgments	
Small-step evaluation	$\mathcal{L}; e \mapsto \mathcal{L}'; e'$
Weak-head reduction	$\Delta \vdash \tau \Downarrow \tau'$
Weak-head normalization	$\Delta \vdash \tau \downarrow \tau'$
Path conversion	$\rho \rightsquigarrow p$

B.4 Static semantics

Type Contexts	$\Delta ::= \cdot$	*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*
Type Isomorphisms	$\Sigma ::= \cdot \mid \Sigma, l{:}\kappa = \tau$	
Term Contexts	$\Gamma ::= \cdot \mid \Gamma, x{:}\sigma$	

Primitive labels and their kinds.

ℓ_{int}	$\star$	*integer label*
$\ell_{\rightarrow}$	$\star \rightarrow \star \rightarrow \star$	*function type creator*

B.4.1 Label well-formedness

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

B.4.2 Label set well-formedness

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

$$\frac{\Delta \vdash \mathcal{L}_1 : \mathrm{Ls} \qquad \Delta \vdash \mathcal{L}_2 : \mathrm{Ls}}{\Delta \vdash \mathcal{L}_1 \cup \mathcal{L}_2 : \mathrm{Ls}}\ \textsf{wfls-core:union} \qquad \frac{}{\Delta \vdash \mathcal{U} : \mathrm{Ls}}\ \textsf{wfls-core:univ} \qquad \frac{s{:}\mathrm{Ls} \in \Delta}{\Delta \vdash s : \mathrm{Ls}}\ \textsf{wfls-core:var}$$

B.4.3 Type well-formedness

$$\frac{\alpha{:}\kappa \in \Delta}{\Delta \vdash \alpha : \kappa}\ \textsf{twf-core:var} \qquad \frac{\alpha{:}\kappa \upharpoonright \mathcal{L} \in \Delta}{\Delta \vdash \alpha : \kappa}\ \textsf{twf-core:var-res} \qquad \frac{\Delta \vdash \tau_1 : \kappa_1 \rightarrow \kappa_2 \qquad \Delta \vdash \tau_2 : \kappa_1}{\Delta \vdash \tau_1 \tau_2 : \kappa_2}\ \textsf{twf-core:app}$$

$$\frac{\Delta, \alpha{:}\kappa_1 \vdash \tau : \kappa_2}{\Delta \vdash \lambda \alpha{:}\kappa_1.\tau : \kappa_1 \rightarrow \kappa_2}\ \textsf{twf-core:abs}$$

$$\frac{\Delta \vdash l : \mathrm{L}(\kappa)}{\Delta \vdash l : \kappa}\ \textsf{twf-core:ltype} \qquad \frac{\Delta \vdash \mathcal{L}_1 : \mathrm{Ls} \qquad \Delta \vdash \tau : \star \to \star \qquad \Delta \vdash \mathcal{L}_2 : \mathrm{Ls}}{\Delta \vdash \mathcal{L}_1 \Rightarrow \tau \restriction \mathcal{L}_2 : \star}\ \textsf{twf-core:map}$$

$$\frac{\Delta \vdash \mathcal{L} : \mathrm{Ls} \qquad \Delta, \alpha{:}\kappa_1 \restriction \mathcal{L} \vdash \tau : \star}{\Delta \vdash \forall \alpha{:}\kappa_1 \restriction \mathcal{L}.\tau : \star}\ \textsf{twf-core:tall} \qquad \frac{\Delta, s{:}\mathrm{Ls} \vdash \tau : \star}{\Delta \vdash \forall s{:}\mathrm{Ls}.\tau : \star}\ \textsf{twf-core:sall}$$

$$\frac{\Delta, \iota{:}\mathrm{L}(\kappa) \vdash \tau : \star}{\Delta \vdash \forall \iota{:}\mathrm{L}(\kappa).\tau : \star}\ \textsf{twf-core:lall}$$

B.4.4 Type label set analysis

$$\frac{\alpha{:}\kappa \in \Delta}{\Delta \vdash \alpha \mid \varnothing}\ \textsf{tan-core:var} \qquad \frac{\alpha{:}\kappa \restriction \mathcal{L} \in \Delta}{\Delta \vdash \alpha \mid \mathcal{L}}\ \textsf{tan-core:var-res} \qquad \frac{\Delta \vdash \tau_1 \mid \mathcal{L}_1 \qquad \Delta \vdash \tau_2 \mid \mathcal{L}_2}{\Delta \vdash \tau_1 \tau_2 \mid \mathcal{L}_1 \cup \mathcal{L}_2}\ \textsf{tan-core:app}$$

$$\frac{\Delta, \alpha{:}\kappa_1 \vdash \tau \mid \mathcal{L}}{\Delta \vdash \lambda \alpha{:}\kappa_1.\tau \mid \mathcal{L}}\ \textsf{tan-core:abs}$$

$$\frac{}{\Delta \vdash l \mid \{l\}}\ \textsf{tan-core:ltype}$$

B.4.5 Label set subsumption

$$\frac{\Delta \vdash \mathcal{L} : \mathrm{Ls}}{\Delta \vdash \mathcal{L} \sqsubseteq \mathcal{L}}\ \textsf{ss-core:refl} \qquad \frac{\Delta \vdash \mathcal{L}_1 \sqsubseteq \mathcal{L}_2 \qquad \Delta \vdash \mathcal{L}_2 \sqsubseteq \mathcal{L}_3}{\Delta \vdash \mathcal{L}_1 \sqsubseteq \mathcal{L}_3}\ \textsf{ss-core:trans}$$

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

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

$$\frac{\Delta \vdash \mathcal{L} : \mathrm{Ls}}{\Delta \vdash \mathcal{L} \sqsubseteq \mathcal{U}}\ \textsf{ss-core:univ}$$

B.4.6 Label set equivalence

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

B.4.7 Type equivalence

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

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

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

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

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

$$\frac{\Delta \vdash \mathcal{L}_{11} = \mathcal{L}_{21} \qquad \Delta \vdash \tau_1 = \tau_2 : \star \to \star \qquad \Delta \vdash \mathcal{L}_{21} = \mathcal{L}_{22}}{\Delta \vdash \mathcal{L}_{11} \Rightarrow \tau_1 \restriction \mathcal{L}_{12} = \mathcal{L}_{21} \Rightarrow \tau_2 \restriction \mathcal{L}_{22} : \star}\ \textsf{teq-core:map-con}$$

$$\frac{\Delta \vdash \mathcal{L}_1 = \mathcal{L}_2 \qquad \Delta, \alpha{:}\kappa \restriction \mathcal{L}_1 \vdash \tau_1 = \tau_2 : \star}{\Delta \vdash \forall \alpha{:}\kappa \restriction \mathcal{L}_1.\tau_1 = \forall \alpha{:}\kappa \restriction \mathcal{L}_2.\tau_2 : \star}\ \textsf{teq-core:tall-con}$$

$$\frac{\Delta, \iota{:}\mathrm{L}(\kappa) \vdash \tau_1 = \tau_2 : \star}{\Delta \vdash \forall \iota{:}\mathrm{L}(\kappa).\tau_1 = \forall \iota{:}\mathrm{L}(\kappa).\tau_2 : \star}\ \textsf{teq-core:lall-con} \qquad \frac{\Delta, s{:}\mathrm{Ls} \vdash \tau_1 = \tau_2 : \star}{\Delta \vdash \forall s{:}\mathrm{L}(\kappa).\tau_1 = \forall s{:}\mathrm{L}(\kappa).\tau_2 : \star}\ \textsf{teq-core:sall-con}$$

B.4.8 Term well-formedness

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

$$\frac{\Delta \vdash \sigma_1 : \star \qquad \Delta; \Gamma, x{:}\sigma_1 \vdash e : \sigma_2 \mid \Sigma}{\Delta; \Gamma \vdash \lambda x{:}\sigma_1.e : \sigma_1 \to \sigma_2 \mid \Sigma}\ \textsf{wf-core:abs}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \sigma_1 \to \sigma_2 \mid \Sigma \qquad \Delta; \Gamma \vdash e_2 : \sigma_1 \mid \Sigma}{\Delta; \Gamma \vdash e_1 e_2 : \sigma_2 \mid \Sigma}\ \textsf{wf-core:app} \qquad \frac{\Delta \vdash \sigma : \star \qquad \Delta; \Gamma, x{:}\sigma \vdash e : \sigma \mid \Sigma}{\Delta; \Gamma \vdash \mathbf{fix}\ x{:}\sigma.e : \sigma \mid \Sigma}\ \textsf{wf-core:fix}$$

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

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

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

$$\frac{\Delta; \Gamma \vdash e : \sigma_1 \mid \Sigma \qquad \Delta \vdash \sigma_1 = \sigma_2 : \star}{\Delta; \Gamma \vdash e : \sigma_2 \mid \Sigma}\ \textsf{wf-core:weak}$$

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

$$\frac{\Delta \vdash \mathcal{L} : \mathrm{Ls} \qquad \Delta \vdash l : \mathrm{L}(\kappa) \qquad \Delta; \Gamma \vdash e : \tau' \langle l : \kappa \restriction \mathcal{L} \rangle \mid \Sigma}{\Delta; \Gamma \vdash \{l \Rightarrow e\} : \{l\} \Rightarrow \tau' \restriction \mathcal{L} \mid \Sigma} \text{ wf-core:env-branch}$$

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

$$\frac{\Delta; \Gamma \vdash e : \mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L}_2 \mid \Sigma \qquad \Delta \vdash \tau : \star \qquad \Delta \vdash \tau \mid \mathcal{L} \qquad \Delta \vdash \mathcal{L}_1 \sqsubseteq \mathcal{L}_2 \qquad \Delta \vdash \mathcal{L} \sqsubseteq \mathcal{L}_1}{\Delta; \Gamma \vdash \mathbf{typecase}\ \tau\ e : \tau' \tau \mid \Sigma} \text{ wf-core:typecase}$$

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

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

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

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

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

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

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

B.5 Dynamic semantics

$$\begin{array}{lll} \textit{Values} & v ::= & \lambda x{:}\sigma.e \\ & \mid & i \\ & \mid & \{\!\{v\}\!\}^{+}_{l=\tau} \\ & \mid & \varnothing \mid \{l \Rightarrow e\} \mid v_1 \bowtie v_2 \\ & \mid & \Lambda\alpha{:}\kappa \restriction \mathcal{L}.e \\ & \mid & \Lambda\iota{:}\mathrm{L}(\kappa).e \\ & \mid & \Lambda s{:}\mathrm{Ls}.e \end{array}$$

$$\begin{array}{lll} \textit{Type paths} & \rho ::= & \bullet \mid \rho\ \tau \\ \textit{Term paths} & p ::= & \bullet \mid p\ [\tau] \end{array}$$

B.5.1 Weak-head reduction for types

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

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

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

B.5.2 Weak-head normalization for types

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

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

B.5.3 Path conversion

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

B.5.4 Computation rules

$$\frac{}{\mathcal{L}; (\lambda x{:}\sigma.e_1)v_2 \mapsto \mathcal{L}; e_1[v_2/x]} \text{ ev-core:abs-beta} \qquad \frac{}{\mathcal{L}; \mathbf{fix}\ x{:}\sigma.e \mapsto \mathcal{L}; e[\mathbf{fix}\ x{:}\sigma.e/x]} \text{ ev-core:fix-beta}$$

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

$$\frac{}{\mathcal{L}; (\Lambda s{:}\mathrm{Ls}.e)[\mathcal{L}] \mapsto \mathcal{L}; e[\mathcal{L}/s]} \text{ ev-core:sabs-beta} \qquad \frac{}{\mathcal{L}; \{\!\{\{\!\{v\}\!\}^+_{l=\tau}\}\!\}^-_{l=\tau} \mapsto \mathcal{L}; v} \text{ ev-core:in-out}$$

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

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

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

$$\frac{\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\ell_{\mathsf{int}}}{\mathcal{L}; \{\!\{i : \tau'\}\!\}^\pm_{l=\tau} \mapsto \mathcal{L}; i} \text{ ev-core:hcolor-int}$$

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

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

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

$$\frac{\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L}_2}{\mathcal{L}; \{\!\{\{\ell^\kappa_i \Rightarrow e'\} : \tau'\}\!\}^{\pm}_{l=\tau} \mapsto \mathcal{L}; \{\ell^k_i \Rightarrow \{\!\{e' : \lambda\alpha{:}\star.\tau'\langle \ell^k_i : \kappa \restriction \mathcal{L}_2\rangle\}\!\}^{\pm}_{l=\tau}\}}\ \text{ev-core:hcolor-sing}$$

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

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

$$\frac{\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\forall s{:}\mathrm{Ls}.\tau'}{\mathcal{L}; \{\!\{\Lambda s{:}\mathrm{Ls}.e : \tau'\}\!\}^{\pm}_{l=\tau} \mapsto \mathcal{L}; \Lambda s{:}\mathrm{Ls}.\{\!\{e : \lambda\alpha{:}\kappa.\tau'\}\!\}^{\pm}_{l=\tau}}\ \text{ev-core:hcolor-sabs}$$

$$\frac{\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\forall \iota{:}\mathrm{L}(\kappa').\tau'}{\mathcal{L}; \{\!\{\Lambda \iota{:}\mathrm{L}(\kappa').e : \tau'\}\!\}^{\pm}_{l=\tau} \mapsto \mathcal{L}; \Lambda \iota{:}\mathrm{L}(\kappa').\{\!\{e : \lambda\alpha{:}\kappa.\tau'\}\!\}^{\pm}_{l=\tau}}\ \text{ev-core:hcolor-labs}$$

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

$$\frac{\vdash \tau \downarrow \rho[\ell^\kappa_i] \qquad \{\ell^\kappa_i \Rightarrow e'\} \in v \qquad \rho \rightsquigarrow p}{\mathcal{L}; \mathbf{typecase}\ \tau\ v \mapsto \mathcal{L}; p[e']}\ \text{ev-core:typecase} \qquad \frac{}{\mathcal{L}; \mathbf{lindex}\ \ell^k_i \mapsto \mathcal{L}; i}\ \text{ev-core:lindex}$$

B.5.5 Congruence rules

$$\frac{\mathcal{L}; e_1 \mapsto \mathcal{L}'; e'_1}{\mathcal{L}; e_1 e_2 \mapsto \mathcal{L}'; e'_1 e_2}\ \text{ev-core:app-con1} \qquad \frac{\mathcal{L}; e \mapsto \mathcal{L}'; e'}{\mathcal{L}; v e \mapsto \mathcal{L}'; v e'}\ \text{ev-core:app-con2}$$

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

$$\frac{\mathcal{L}; e_1 \mapsto \mathcal{L}'; e'_1}{\mathcal{L}; e_1 \bowtie e_2 \mapsto \mathcal{L}'; e'_1 \bowtie e_2}\ \text{ev-core:join-con1} \qquad \frac{\mathcal{L}; e_2 \mapsto \mathcal{L}'; e'_2}{\mathcal{L}; v \bowtie e \mapsto \mathcal{L}'; v \bowtie e'_2}\ \text{ev-core:join-con2}$$

$$\frac{\mathcal{L}; e \mapsto \mathcal{L}'; e'}{\mathcal{L}; \mathbf{typecase}\ \tau\ e \mapsto \mathcal{L}'; \mathbf{typecase}\ \tau\ e'}\ \text{ev-core:typecase-con} \qquad \frac{\mathcal{L}; e_1 \mapsto \mathcal{L}'; e'_1}{\mathcal{L}; e_1[\tau] \mapsto \mathcal{L}'; e'_1[\tau]}\ \text{ev-core:tapp-con}$$

$$\frac{\mathcal{L}; e_1 \mapsto \mathcal{L}'; e'_1}{\mathcal{L}; e_1[\hat{l}] \mapsto \mathcal{L}'; e'_1[\hat{l}]}\ \text{ev-core:lapp-con} \qquad \frac{\mathcal{L}; e_1 \mapsto \mathcal{L}'; e'_1}{\mathcal{L}; e_1[\mathcal{L}] \mapsto \mathcal{L}'; e'_1[\mathcal{L}]}\ \text{ev-core:sapp-con}$$

C Properties

C.1 Erasure

In this section we prove that the coercion mechanism does not have an operational effect, that is, operationally the core $\lambda_{\mathcal{L}}$ and a language where all coercions are erased are equivalent.

C.1.1 Target Syntax

In the following we give the syntax of core $\lambda_{\mathcal{L}}$ without the coercion mechanism. The dynamic and static semantics are exactly what one would expect—a restriction of the semantics of the core $\lambda_{\mathcal{L}}$—and not presented.

Kinds	κ	$::=$	$\star$	*normal types*
		$\mid$	$\kappa_1 \to \kappa_2$	*function kinds*
Labels	l	$::=$	ℓ_i^{κ}	*constants*
		$\mid$	ι	*variables*
label sets	$\mathcal{L}$	$::=$	$\varnothing$	*empty*
		$\mid$	$\{l\}$	*singleton*
		$\mid$	s	*variable*
		$\mid$	$\mathcal{L}_1 \cup \mathcal{L}_2$	*join*
		$\mid$	$\mathcal{U}$	*universe*
Types	σ, τ	$::=$	$\alpha \mid \lambda\alpha{:}\kappa.\tau \mid \tau_1\tau_2$	*λ-calculus*
		$\mid$	$\forall\iota{:}\mathrm{L}(\kappa).\tau$	*label quantification*
		$\mid$	$\forall\alpha{:}\kappa \restriction \mathcal{L}.\tau$	*type quantification*
		$\mid$	$\forall s{:}\mathrm{Ls}.\tau$	*label set quantification*
		$\mid$	l	*label coercion*
		$\mid$	$\mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L}_2$	*branch types*
Terms	e	$::=$	$x \mid \lambda x.e \mid e_1 e_2$	*λ-calculus*
		$\mid$	$\mathbf{fix}\ x.e$	*recursion*
		$\mid$	i	*integers*
		$\mid$	$\mathbf{new}\ \iota\ \mathbf{in}\ e$	*label creation*
		$\mid$	$\mathbf{typecase}\ \tau\ e$	*type case analysis*
		$\mid$	$\mathbf{lindex}\ l$	*label analysis*
		$\mid$	$\Lambda\alpha{:}\kappa \restriction \mathcal{L}.e \mid e[\tau]$	*type polymorphism*
		$\mid$	$\Lambda\iota{:}\mathrm{L}(\kappa).e \mid e[l]$	*label polymorphism*
		$\mid$	$\Lambda s{:}\mathrm{Ls}.e \mid e[\mathcal{L}]$	*label set polymorphism*
		$\mid$	$\varnothing \mid \{l \Rightarrow e\} \mid e_1 \bowtie e_2$	*branches*

C.1.2 Erasure Semantics

Here we define the erasure relation, that relates terms of core $\lambda_{\mathcal{L}}$ to terms of the target syntax.

$$
\begin{array}{lcl}
|x| & = & x \\
|\lambda x{:}\sigma.e| & = & \lambda x{:}\sigma.|e| \\
|e_1 e_2| & = & |e_1||e_2| \\
|\textbf{fix } x{:}\sigma.e| & = & \textbf{fix } x{:}\sigma.|e| \\
|i| & = & i \\
|\textbf{new } \iota{:}\kappa = \tau \textbf{ in } e| & = & \textbf{new } \iota \textbf{ in } |e| \\
|\{\!\{e\}\!\}^{\pm}_{l=\tau}| & = & |e| \\
|\{\!\{e : \tau\}\!\}^{\pm}_{l=\tau_2}| & = & |e| \\
|\textbf{typecase } \tau \ e| & = & \textbf{typecase } \tau \ |e| \\
|\textbf{lindex } l| & = & \textbf{lindex } l \\
|\Lambda\alpha{:}\kappa \upharpoonright \mathcal{L}.e| & = & \Lambda\alpha{:}\kappa \upharpoonright \mathcal{L}.|e| \\
|e[\tau]| & = & |e|[\tau] \\
|\Lambda\iota{:}\mathrm{L}(\kappa).e| & = & \Lambda\iota{:}\mathrm{L}(\kappa).|e| \\
|e[l]| & = & |e|[l] \\
|\Lambda s{:}\mathrm{Ls}.e| & = & \Lambda s{:}\mathrm{Ls}.|e| \\
|e[\mathcal{L}]| & = & |e|[\mathcal{L}] \\
|\varnothing| & = & \varnothing \\
|\{l \Rightarrow e\}| & = & \{l \Rightarrow |e|\} \\
|e_1 \bowtie e_2| & = & |e_1| \bowtie |e_2|
\end{array}
$$

C.1.3 Source Calculus Metric

It turns out that we need to define a metric relation on terms of core $\lambda_{\mathcal{L}}$. The definition of this relation is as follows:

$$
\begin{array}{lcl}
size(x) & = & 1 \\
size(\lambda x{:}\sigma.e) & = & 1 + size(e) \\
size(e_1 e_2) & = & size(e_1) + size(e_2) \\
size(\textbf{fix } x{:}\sigma.e) & = & 1 + size(e) \\
size(i) & = & 1 \\
size(\textbf{new } \iota{:}\kappa = \tau \textbf{ in } e) & = & 1 + size(e) \\
size(\{\!\{e\}\!\}^{\pm}_{l=\tau}) & = & size(e) + 1 \\
size(\{\!\{e : \tau\}\!\}^{\pm}_{l=\tau_2}) & = & size(e) \\
size(\textbf{typecase } \tau \ e) & = & 1 + size(e) \\
size(\textbf{lindex } l) & = & 1 \\
size(\Lambda\alpha{:}\kappa \upharpoonright \mathcal{L}.e) & = & 1 + size(e) \\
size(e[\tau]) & = & 1 + size(e) \\
size(\Lambda\iota{:}\mathrm{L}(\kappa).e) & = & 1 + size(e) \\
size(e[l]) & = & 1 + size(e) \\
size(\Lambda s{:}\mathrm{Ls}.e) & = & 1 + size(e) \\
size(e[\mathcal{L}]) & = & 1 + size(e) \\
size(\varnothing) & = & 1 \\
size(\{l \Rightarrow e\}) & = & 1 + size(e) \\
size(e_1 \bowtie e_2) & = & size(e_1) + size(e_2)
\end{array}
$$

C.1.4 Normalization Relation

Finally, we need to define a relation between paths and types, with the properties shown below. Later we will have to prove that all pairs of paths and types that are "compatible", that is, we can plug in the type in the hole of path are in this relation.

$$
\begin{array}{lcll}
(\rho,\tau)\in Norm & \Leftrightarrow & \vdash \tau:\kappa & N1 \quad \text{and} \\
 & & \alpha{:}\kappa \vdash \rho[\alpha]:\star & N2 \\
 & & \text{and} & \\
 & & \alpha{:}\kappa \vdash \rho[\tau]\Downarrow_* \rho'[\alpha] \text{ and } (\rho',\tau)\in Norm & N3a \\
 & & \alpha{:}\kappa \vdash \rho[\tau]\Downarrow_* \rho'[\ell_i^\kappa] & N3b \quad \text{or} \\
 & & \alpha{:}\kappa \vdash \rho[\tau]\Downarrow_* \forall\phi & N3c \quad \text{or} \\
 & & \alpha{:}\kappa \vdash \rho[\tau]\Downarrow_* \mathcal{L}\Rightarrow\tau'\restriction\mathcal{L}' & N3d \quad \text{or}
\end{array}
$$

Lemma C.1. *If $\mathcal{L};e\mapsto\mathcal{L}';e'$ then $\mathcal{L};|e|\mapsto^*\mathcal{L}';|e'|$.*

Proof. Straightforward induction on the evaluation relation of the source calculus. Because of the coercions we need to allow for zero steps in the target calculus. □

Corollary C.2 (Adequacy). *If $\mathcal{L};e\mapsto^*\mathcal{L}';v$ then $\mathcal{L};|e|\mapsto^*\mathcal{L}';|v|$.*

Proof. By induction on the number of steps of the evaluation on the source calculus. Actually we prove that if $\mathcal{L};e\mapsto^n\mathcal{L}';v$ then $\mathcal{L};|e|\mapsto^*\mathcal{L}';|v|$. For $n=0$ we have that $e=v$, $\mathcal{L}=\mathcal{L}'$, and $\mathcal{L};|v|\mapsto^*\mathcal{L}';|v|$. Suppose now $n\geq 1$ and take $\mathcal{L};e\mapsto^n\mathcal{L}'';v$, where $\mathcal{L};e\mapsto\mathcal{L}';e'$ and $\mathcal{L}';e'\mapsto^{n-1}\mathcal{L}'';v$. Then by induction hypothesis $\mathcal{L}';|e'|\mapsto^*\mathcal{L}'';|v|$. And by Lemma C.1 we get $\mathcal{L};|e|\mapsto^*\mathcal{L}';|e'|$. Combining we get $\mathcal{L};|e|\mapsto^*\mathcal{L}';|v|$. □

Lemma C.3. *If $\Delta,\alpha{:}\kappa_1\vdash\tau\Downarrow\tau'$ and $\Delta,\alpha{:}\kappa_1\vdash\tau:\kappa$ and $\Delta\vdash\tau_1:\kappa_1$, then $\Delta\vdash\tau[\tau_1/\alpha]\Downarrow\tau'[\tau_1/\alpha]$.*

Proof. By induction on the weak head reduction relation. Consider the case for whr-core:abs-beta. Then $\Delta,\alpha{:}\kappa_1\vdash(\lambda\beta{:}\kappa_2.\tau_2)\tau_3\Downarrow\tau[\tau_3/\beta]_2$. But $(\lambda\beta{:}\kappa_2.\tau_2)\tau_3[\tau_1/\alpha]=(\lambda\beta{:}\kappa_2.\tau_2[\tau_1/\alpha])\tau_3[\tau_1/\alpha]$, and this reduces to $\tau_2[\tau_3/\beta][\tau_1/\alpha]$. The cases for whr-core:abs-con and whr-core:app-con follow easily from the induction hypothesis. □

Corollary C.4. *If $\Delta,\alpha{:}\kappa_1\vdash\tau\Downarrow_n\tau'$ and $\Delta,\alpha{:}\kappa_1\vdash\tau:\kappa$ and $\Delta\vdash\tau_1:\kappa_1$, then $\Delta\vdash\tau[\tau_1/\alpha]\Downarrow_n\tau'[\tau_1/\alpha]$.*

Proof. By induction on n. The case for $n=0$ is trivial. The inductive step follows easily using Lemma C.3. □

Lemma C.5. *If $\vdash\tau:\kappa$ and $\alpha{:}\kappa\vdash\rho[\alpha]:\star$ and $\alpha{:}\kappa\vdash\rho[\tau/\alpha][\tau]\Downarrow_n\tau'$ and $\alpha{:}\kappa\vdash\tau'\not\Downarrow$ then $(\rho,\tau)\in Norm$.*

Proof. By induction on n. The properties $N1$ and $N2$ are trivially satisfied for all n by our assumptions.

- Case $n=0$. Then we have that $\alpha{:}\kappa\vdash\rho[\tau/\alpha][\tau]\not\Downarrow$. Since this type cannot reduce further and τ is closed, it must be that either $\tau=\rho'[\ell_i^k]$, or $\tau=\forall\phi$, or $\tau=\mathcal{L}_1\Rightarrow\tau'\restriction\mathcal{L}_2$. In the first case $\rho[\tau]=\rho[\rho'[\ell_i^k]]\Downarrow_0\rho''[\ell_i^k]$, which means that $N3b$ is satisfied, and therefore $(\rho,\tau)\in Norm$. In the second case $\rho[\forall\phi]\Downarrow_0\rho[\forall\phi]=\forall\phi$, because we know that $\forall\phi$ is always of kind $\star$ and therefore ρ must be just a hole in this case. But then $N3c$ is satisfied and therefore $(\rho,\tau)\in Norm$. In the third case $\rho[\mathcal{L}_1\Rightarrow\tau'\restriction\mathcal{L}_2]\Downarrow_0\mathcal{L}_1\Rightarrow\tau'\restriction\mathcal{L}_2$, because again map types are of kind $\star$. Therefore $N3d$ is satisfied and $(\rho,\tau)\in Norm$.
- Case $n\geq 1$. Now consider how the term $\rho[\tau]$ reduces, and suppose that it reaches weak head normal form in k steps. If must be that $k\leq n$, because if this was not the case, we would contradict Corollary C.4. Formally $\alpha{:}\kappa\vdash\rho[\tau]\Downarrow_k\tau''$, with $\alpha{:}\kappa\vdash\tau''\not\Downarrow$. If $\tau''=\rho[\ell_i^k]$ or $\tau''=\forall\phi$ or $\tau''=\mathcal{L}_1\Rightarrow\tau'\restriction\mathcal{L}_2$, then we are done, because we can use Corollary C.4 to get the necessary conditions. Suppose now that $\tau''=\rho'[\alpha]$, that is, $\alpha{:}\kappa\vdash\rho[\tau]\Downarrow_k\rho'[\alpha]$. Consider cases for k.
  - Case $k=0$. This can't happen because in this case $\rho=\rho'$ and $\alpha=\tau$, but $\vdash\tau:\star$, which means that τ cannot contain the free variable α.
  - Case $k=n$. This means by Corollary C.4 that $\alpha{:}\kappa\vdash\rho[\tau/\alpha][\tau]\Downarrow_n\rho'[\tau/\alpha][\tau]$, and $\tau'=\rho'[\tau/\alpha][\tau]k$. Now since $\alpha{:}\kappa\vdash\tau'\not\Downarrow$ by our assumptions, it must be that $\tau=\rho'[\ell_i^k]$, or $\tau=\forall\phi$, or $\tau=\mathcal{L}_1\Rightarrow\tau'\restriction\mathcal{L}_2$, which also means that $\alpha{:}\kappa\vdash\rho[\tau/\alpha][\tau]\not\Downarrow$. But this means that $n=0$, which cannot happen in the case we are examining.

– Case $1 \leq k \leq n-1$. Again by Corollary C.4 we get that $\alpha{:}\kappa \vdash \rho[\tau/\alpha][\tau] \Downarrow_k \rho'[\tau/\alpha][\tau]$. But then, by our asumptions we know that $\alpha{:}\kappa \vdash \rho'[\tau/\alpha][\tau] \Downarrow_m \tau'$, where $k+m=n$. But then, by induction hypothesis $(\rho', \tau) \in Norm$ and we knew that $\alpha{:}\kappa \vdash \rho[\tau] \Downarrow_k \rho'[\alpha]$, which means that property $N3a$ is satisfied and therefore $(\rho, \tau) \in Norm$.

□

Corollary C.6. *If* $\vdash \tau : \kappa$ *and* $\alpha{:}\kappa \vdash \rho[\alpha] : \star$*, then* $(\rho, \tau) \in Norm$.

Proof. Just observe that weak head reduction $\Downarrow_*$ is deterministic and strongly normalizing, so by Lemma C.5 all paths and types satisfying our assumptions are in $Norm$. □

Lemma C.7 (Strong normalization for coercions of values). *If* $.; . \vdash \{\!\{v : \tau'\}\!\}^{\pm}_{l=\tau} : \sigma \,|\, \Sigma$*, where* $\ell_{\mathsf{int}}, \ell_{\rightarrow} \notin dom(\Sigma)$ *and* $\mathcal{L} = dom(\Sigma) \cup \ell_{\mathsf{int}} \cup \ell_{\rightarrow}$*, then* $\mathcal{L}; \{\!\{v : \tau'\}\!\}^{\pm}_{l=\tau} \mapsto^* \mathcal{L}'; v'$.

Proof. By induction on the metric $size(v)$. The induction hypothesis will say that every coercion of a smaller-size value terminates. We proceed by case analysis on the first step of the evaluation. The cases for ev-core:hcolor-int, ev-core:hcolor-empty, ev-core:hcolor-abs1, ev-core:hcolor-abs2, ev-core:hcolor-branch, ev-core:hcolor-tabs, ev-core:hcolor-labs, ev-core:hcolor-sabs are immediate because the term becomes value in the first step. Here are the rest of the cases.

- Case ev-core:hcolor-join. Then $\mathcal{L}; \{\!\{v_1 \bowtie v_2 : \tau'\}\!\}^{\pm}_{l=\tau} \mapsto \mathcal{L}; \{\!\{v_1 : \lambda\alpha{:}\kappa.\mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L}\}\!\}^{\pm}_{l=\tau} \bowtie \{\!\{v_2 : \lambda\alpha{:}\kappa.\mathcal{L}_2 \Rightarrow \tau' \restriction \mathcal{L}\}\!\}^{\pm}_{l=\tau}$. But note that $size(v_1) \langle size(v_1 \bowtie v_2)$, and similarly $size(v_2) \langle size(v_1 \bowtie v_2)$. The result follows by induction hypothesis and the transitivity of the evaluation relation.

- Case ev-core:hcolor-color. In this case $\mathcal{L}; \{\!\{\{\!\{v\}\!\}^{+}_{l_1=\tau_1} : \tau'\}\!\}^{\pm}_{l_2=\tau_2} \mapsto \mathcal{L}; \{\!\{\{\!\{v : \lambda\alpha{:}\kappa.\rho[\tau_1]\}\!\}^{\pm}_{l_2=\tau_2}\}\!\}^{+}_{l_1=\tau_1}$. Note that $size(\{\!\{v\}\!\}^{+}_{l_1=\tau_1}) = size(v) + 1$, while $size(\{\!\{v : \lambda\alpha{:}\kappa.\rho[\tau_1]\}\!\}^{\pm}_{l_2=\tau_2}) = size(v)$. Therefore, by induction hypothesis $\mathcal{L}; \{\!\{v : \lambda\alpha{:}\kappa.\rho[\tau_1]\}\!\}^{\pm}_{l_2=\tau_2} \mapsto^* \mathcal{L}'; w$, where w is a value. Now if we were examining an $+$ case, trivially $\{\!\{w\}\!\}^{+}_{l_1=\tau_1}$ is a value. If we were examining a $-$ case then we have that $.; . \vdash \{\!\{w : \tau'\}\!\}^{-}_{l=\tau} : \sigma \,|\, \Sigma$ by preservation theorem (see later). Also by the inversion lemma (see later) on typing we get that $; \vdash w : \rho[\ell^k_i] \,|\, \Sigma$, where ℓ^k_i cannot be $\ell_{\mathsf{int}}, \ell_{\rightarrow}$, because of our assumptions. But then, by the canonical forms lemma (see later) there exists a value w', such that $w = \{\!\{w'\}\!\}^{+}_{l_1=\tau_1}$ and by applying one time the rule ev-core:in-out we are done.

- Case ev-core:hcolor-base-in. Here $\mathcal{L}; \{\!\{v : \tau'\}\!\}^{+}_{l=\tau} \mapsto \mathcal{L}; \{\!\{\{\!\{v : \lambda\alpha{:}\kappa.\rho[\tau]\}\!\}^{+}_{l=\tau}\}\!\}^{+}_{l=\tau}$. Now the only rule that can be applied here is the congruence rule, because by preservation the second term is well formed and by progress it can take a step since it is not a value. Therefore it must be that the second term reduced to an expression e'. We have two cases.

  – Suppose that one of the rules except ev-core:hcolor-base or ev-core:hcolor-base-out was appplied. Then it is easy to verify that after one or two steps we either reach a value or we will be able to apply the induction hypothesis to see that the term normalizes to a value.

  – Suppose that the rule applied was ev-core:hcolor-base again. Then $\vdash \lambda\alpha{:}\kappa.\rho[\tau] \Downarrow \lambda\alpha{:}\kappa.\rho'[\alpha]$ and by inversion (see later) $\alpha{:}\kappa \vdash \rho[\tau] \Downarrow_* \rho'[\alpha]$. But now by inversion once more $\alpha{:}\kappa \vdash \rho[\alpha] : \star$ and $\vdash \tau : \kappa$. But these are the conditions we need for Corollary C.6 to see that $(\rho, \tau) \in Norm$. This means that the rule ev-core:hcolor-base can be applied after itself only a finite number of times before the type normalized to some other form; since $Norm$ is the smallest relation closed under its definition. Eventually after executing the same rules we will jump to another rule and we will be able to apply the induction hypothesis.

  – Suppose that the rule applied was ev-core:hcolor-base-out. The argument is similar to the argument for the case when the rule is independently applied. See below.

- Case ev-core:hcolor-base-out. Here $\mathcal{L}; \{\!|v : \tau'|\!\}^{-}_{l=\tau} \mapsto \mathcal{L}; \{\!|\{\!|v : \lambda\alpha{:}\kappa.\rho[l]|\!\}^{-}_{l=\tau}|\!\}^{-}_{l=\tau}$. Observe that it cannot be that $\vdash \lambda\alpha{:}\kappa.\rho[l] \Downarrow \lambda\alpha{:}\kappa.\rho'[\alpha]$, which means that one of the other rules will apply and in one or two steps we either reach a value or we can apply the induction hypothesis which will give us that we will reduce to a value.

□

Lemma C.8. *If* $; \vdash e : \sigma \mid \Sigma$, $|e| = v$ $\ell_{\mathsf{int}}, \ell_{\rightarrow} \notin dom(\Sigma)$, $\mathcal{L} = dom(\Sigma) \cup \ell_{\mathsf{int}} \cup \ell_{\rightarrow}$, *then* $\mathcal{L}; e \mapsto^* \mathcal{L}'; v'$, *with* $|v'| = v$.

Proof. By induction on the erasure relation; observe how values v can occure from erasures or expressions e. The only interesting cases are the following:

- Case $e = \{\!|e'|\!\}^{\pm}_{l=\tau}$ given that $|e'| = v$. By inversion e' is well formed and by induction hypothesis $\mathcal{L}; e' \mapsto^* \mathcal{L}'; v$, which implies $\mathcal{L}; \{\!|e'|\!\}^{\pm}_{l=\tau} \mapsto^* \mathcal{L}'; \{\!|v|\!\}^{\pm}_{l=\tau}$. If we were examining the $+$, case the last term is a value and we are done. If we were examining the case for $\{\!|v|\!\}^{-}_{l=\tau}$, then by applying preservation, inversion on typing and the canonical forms lemma we will get that there exists a v' value, such that $v = \{\!|v'|\!\}^{+}_{l=\tau}$, and observe that these two erase to the same value. Then by applying ev-core:in-out we are done.
- Case $e = \{\!|e' : \tau'|\!\}^{\pm}_{l=\tau}$, given that $v = |e'|$. By inversion, e' is well formed and by induction hypothesis $\mathcal{L}; e' \mapsto^* \mathcal{L}'; v$, which implies that $\mathcal{L}; \{\!|e' : \tau'|\!\}^{\pm}_{l=\tau} \mapsto^* \mathcal{L}'; \{\!|v : \tau'|\!\}^{\pm}_{l=\tau}$, and now by Lemma C.7 we are done.

□

Lemma C.9. *If* $\mathcal{L}; |e| \mapsto \mathcal{L}'; e_1$, *then there exists an* e_2, *such that* $e_1 = |e_2|$ *and* $\mathcal{L}; e \mapsto^* \mathcal{L}'; e_2$.

Proof. Easy induction on the erasure relation, appealing to Lemma C.8 in the case for application. □

Theorem C.10 (Full Abstraction). *If* $\mathcal{L}; |e| \mapsto^* \mathcal{L}'; |v|$, *then* $\exists v'$, *such that* $|v'| = v$ *and* $\mathcal{L}; e \mapsto^* \mathcal{L}'; v'$.

Proof. The proof is by induction on the number of steps of the evaluation of the erased term, $\mathcal{L}; |e| \mapsto^n \mathcal{L}'; |v|$.

- Case $n = 0$. Then $|e| = v$ and by Lemma C.7 $\mathcal{L}; e \mapsto^* \mathcal{L}'; v'$ with $|v'| = v$.
- Case $n \geq 1$. Then we have that $\mathcal{L}; |e| \mapsto \mathcal{L}''; e_1$, and then $\mathcal{L}''; e_1 \mapsto^* \mathcal{L}'; v$. But by Lemma C.8 and Lemma C.9, there exists an e_2 such that $|e_2| = e_1$ and $\mathcal{L}; e \mapsto^* \mathcal{L}''; e_2$. But then $\mathcal{L}''; |e_2| \mapsto^* \mathcal{L}'; v$ in fewer steps, and by induction hypothesis there exists a v' such that $|v'| = v$ and $\mathcal{L}''; e_2 \mapsto^* \mathcal{L}'; v'$. But now by transitivity we get that $\mathcal{L}; e \mapsto^* \mathcal{L}'; v'$ and we are done.

□

C.2 Inversion on Typing

Lemma C.11.

1. *If* $\Delta; \Gamma \vdash \lambda x{:}\sigma.e : \sigma_1 \rightarrow \sigma_2 \mid \Sigma$ *then* $\Delta \vdash \sigma = \sigma_1 : \star$ *and* $\Delta; \Gamma, x{:}\sigma \vdash e : \sigma_2 \mid \Sigma$.
2. *If* $\Delta; \Gamma \vdash \Lambda\alpha{:}\kappa \restriction \mathcal{L}.e : \forall\alpha{:}\kappa' \restriction \mathcal{L}'.\sigma \mid \Sigma$ *then* $\kappa = \kappa'$, $\Delta \vdash \mathcal{L} = \mathcal{L}'$ *and* $\Delta, \alpha{:}\kappa \restriction \mathcal{L}; \Gamma \vdash e : \sigma \mid \Sigma$.
3. *If* $\Delta; \Gamma \vdash \Lambda\iota{:}\mathrm{L}(\kappa).e : \forall\iota{:}\mathrm{L}(\kappa').\sigma \mid \Sigma$ *then* $\kappa = \kappa'$, $\Delta, \iota{:}\mathrm{L}(\kappa); \Gamma \vdash e : \sigma \mid \Sigma$.
4. *If* $\Delta; \Gamma \vdash \Lambda s{:}\mathrm{Ls}.e : \forall s{:}\mathrm{Ls}.\sigma \mid \Sigma$ *then* $\Delta, s{:}\mathrm{Ls}; \Gamma \vdash e : \sigma \mid \Sigma$.
5. *If* $\Delta; \Gamma \vdash \ldots \bowtie \{l \Rightarrow e\} \bowtie \ldots : \mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L}_2 \mid \Sigma$ *then* $\Delta \vdash \{l\} \sqsubseteq \mathcal{L}_1$, $\Delta \vdash l : \mathrm{L}(\kappa)$ *and* $\Delta; \Gamma \vdash e : \tau' \langle l : \kappa \restriction \mathcal{L}_2 \rangle \mid \Sigma$.
6. *If* $\Delta; \Gamma \vdash \{l \Rightarrow e\} : \mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L}_2 \mid \Sigma$ *then* $\Delta \vdash \{l\} = \mathcal{L}_1$, $\Delta \vdash l : \mathrm{L}(\kappa)$ *and* $\Delta; \Gamma \vdash e : \tau' \langle l : \kappa \restriction \mathcal{L}_2 \rangle \mid \Sigma$.

7. *If* $\Delta;\Gamma \vdash \varnothing : \mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L}_2 \mid \Sigma$ *then* $\Delta \vdash \mathcal{L}_1 = \varnothing$.

Proof. Straightforward induction over the structure of the typing derivations. □

Lemma C.12 (Extra inversions). *The following hold:*

1. *If* $\Delta;\Gamma \vdash \{\!\{e\}\!\}^{+}_{l=\tau} : \rho[l] \mid \Sigma$ *then* $\Delta;\Gamma \vdash e : \rho[\tau] \mid \Sigma$.
2. *If* $\Delta;\Gamma \vdash \{\!\{e\}\!\}^{-}_{l=\tau} : \rho[\tau] \mid \Sigma$ *then* $\Delta;\Gamma \vdash e : \rho[l] \mid \Sigma$.
3. *If* $\Delta;\Gamma \vdash \{\!\{e : \tau'\}\!\}^{+}_{l=\tau} : \tau'\ l \mid \Sigma$ *then* $\Delta;\Gamma \vdash e : \tau'\ \tau \mid \Sigma$.
4. *If* $\Delta;\Gamma \vdash \{\!\{e : \tau'\}\!\}^{-}_{l=\tau} : \tau'\ \tau \mid \Sigma$ *then* $\Delta;\Gamma \vdash e : \tau'\ l \mid \Sigma$.
5. *If* $\Delta;\Gamma \vdash e_1 \bowtie e_2 : \mathcal{L}_0 \Rightarrow \tau' \restriction \mathcal{L} \mid \Sigma$ *then* $\exists \mathcal{L}_1, \mathcal{L}_2$, *such that* $\Delta \vdash \mathcal{L}_0 = \mathcal{L}_1 \cup \mathcal{L}_2$ *and* $\Delta;\Gamma \vdash e_1 : \mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L} \mid \Sigma$, $\Delta;\Gamma \vdash e_2 : \mathcal{L}_2 \Rightarrow \tau' \restriction \mathcal{L} \mid \Sigma$.

Proof. Straightforward induction over the structure of the typing derivations. □

C.3 Auxilliary Lemmas

Lemma C.13 (Type substitution preserves kinds). *If* $\Delta, \alpha{:}\kappa_1 \vdash \tau : \kappa_2$ *and* $\Delta \vdash \tau_1 : \kappa_1$ *then* $\Delta \vdash \tau[\tau_1/\alpha] : \kappa_2$.

Proof. Straightforward induction on kinding derivations using an easy weakening and permutation lemma for type contexts, which we ommit. □

Lemma C.14 (Type reduction preserves kinds). *If* $\Delta \vdash \tau : \kappa$ *and* $\Delta \vdash \tau \Downarrow \tau'$ *then* $\Delta \vdash \tau' : \kappa$.

Proof. Easy induction on kinding derivations, appealing to Lemma C.13 where necessary. □

Lemma C.15 (Type substitution preserves label set analysis). *If* $\Delta, \alpha{:}\kappa_1 \vdash \tau_1 \mid \mathcal{L}_1$ *and* $\Delta \vdash \tau_2 \mid \mathcal{L}_2$ *then* $\Delta \vdash \tau_1[\tau_2/\alpha] \mid \mathcal{L}'$, *with* $\Delta \vdash \mathcal{L}' \sqsubseteq \mathcal{L}_1 \cup \mathcal{L}_2$.

Proof. By induction on the label set analysis derivation $\Delta, \alpha{:}\kappa_1 \vdash \tau_1 \mid \mathcal{L}_1$. We assume the weakening and permutation properties for Δ with respect to the label set analysis.

- Cases tan-core:var and tan-core:var-res. We have that $\Delta, \alpha : \kappa \vdash \beta \mid \mathcal{L}_1$, given that either $\mathcal{L}_1 = \varnothing$ and $\beta{:}\kappa' \in \Delta, \alpha : \kappa$ or $\beta : \kappa' \restriction \mathcal{L}_1 \in \Delta, \alpha : \kappa$. If $\beta = \alpha$, then $\mathcal{L}_1 = \varnothing$ and $\beta[\tau_2/\alpha] = \tau_2$, with $\Delta \vdash \tau_2 \mid \mathcal{L}_2$ and $\Delta \vdash \mathcal{L}_2 \sqsubseteq \varnothing \cup \mathcal{L}_2$. If $\beta \neq \alpha$, then $\beta[\tau_2/\alpha] = \beta$ and by weakening $\Delta \vdash \beta \mid \mathcal{L}_1$ and we know that $\Delta \vdash \mathcal{L}_1 \sqsubseteq \mathcal{L}_1 \cup \mathcal{L}_2$.
- Case tan-core:app. Straightforward application of the induction hypothesis.
- Case tan-core:ltype. We have that $\Delta \vdash l \mid \{l\}$. The substitution will leave l as it is and it is easy to verify that $\Delta \vdash \{l\} \sqsubseteq \{l\} \cup \mathcal{L}_2$.
- Case tan-core:abs. Here $\Delta, \alpha{:}\kappa \vdash \lambda\beta{:}\kappa_1.\tau \mid \mathcal{L}_1$, given that $\Delta, \alpha{:}\kappa, \beta{:}\kappa_1 \vdash \tau \mid \mathcal{L}_1$. By our assumptions $\Delta \vdash \tau_2 \mid \mathcal{L}_2$ and by weakening $\Delta, \beta{:}\kappa_1 \vdash \tau_2 \mid \mathcal{L}_2$. By permutation $\Delta, \beta{:}\kappa_1, \alpha{:}\kappa \vdash \tau \mid \mathcal{L}_1$. Then by induction hypothesis $\Delta, \beta{:}\kappa_1 \vdash \tau[\tau_2/\alpha] \mid \mathcal{L}'$, where $\Delta, \beta{:}\kappa_1 \vdash \mathcal{L}' \sqsubseteq \mathcal{L}_1 \cup \mathcal{L}_2$. Now, by tan-core:abs we get $\Delta \vdash \lambda\beta{:}\kappa_1.\tau[\tau_2/\alpha] \mid \mathcal{L}'$ which is equivalent to $\Delta \vdash \lambda\beta{:}\kappa_1.\tau_2[\tau_2/\alpha] \mid \mathcal{L}'$ and by weakening for the label set subset relation $\Delta \vdash \mathcal{L}' \sqsubseteq \mathcal{L}_1 \cup \mathcal{L}_2$.

□

Lemma C.16 (Type reduction preserves label set analysis). *If* $\Delta \vdash \tau \mid \mathcal{L}$ *and* $\Delta \vdash \tau \Downarrow \tau'$ *then* $\Delta \vdash \tau' \mid \mathcal{L}'$, *with* $\Delta \vdash \mathcal{L}' \sqsubseteq \mathcal{L}$.

Proof. By induction on the label set analysis derivations. The cases for tan-core:var, tan-core:ltype are trivial because the type cannot take a step. The interesting cases ase tan-core:abs and tan-core:app.

- Case tan-core:abs. We have $\Delta \vdash \lambda\alpha{:}\kappa_1.\tau \,|\, \mathcal{L}$, given that $\Delta, \alpha{:}\kappa_1 \vdash \tau \,|\, \mathcal{L}$. There is one chance for evaluation, using the rule whr-core:abs-con, that is $\Delta, \alpha{:}\kappa_1 \vdash \tau \Downarrow \tau'$. By induction hypothesis $\Delta, \alpha{:}\kappa_1 \vdash \tau' \,|\, \mathcal{L}'$, with $\Delta, \alpha{:}\kappa_1 \vdash \mathcal{L}' \sqsubseteq \mathcal{L}$. Finally by tan-core:abs we get that $\Delta \vdash \lambda\alpha{:}\kappa_1.\tau' \,|\, \mathcal{L}'$ and we are done.
- Case tan-core:app. Here we have that $\Delta \vdash \tau_1\tau_2 \,|\, \mathcal{L}_1 \cup \mathcal{L}_2$, given that $\Delta \vdash \tau_1 \,|\, \mathcal{L}_1$, $\Delta \vdash \tau_2 \,|\, \mathcal{L}_2$. There are two cases for reduction.
  - Case whr-core:app-con. Then $\Delta \vdash \tau_1 \Downarrow \tau_1'$, and by induction hypothesis $\Delta \vdash \tau_1' \,|\, \mathcal{L}_1'$, with $\Delta \vdash \mathcal{L}_1' \sqsubseteq \mathcal{L}_1$, which implies that $\Delta \vdash \mathcal{L}_1' \cup \mathcal{L}_2 \sqsubseteq \mathcal{L}_1 \cup \mathcal{L}_2$, and by tan-core:app we have $\Delta \vdash \tau_1'\tau_2 \,|\, \mathcal{L}_1' \cup \mathcal{L}_2$.
  - Case whr-core:abs-beta. Then $\Delta \vdash (\lambda\alpha{:}\kappa.\tau_1')\tau_2 \Downarrow \tau_1'[\tau_2/\alpha]$. We know that $\Delta \vdash \lambda\alpha{:}\kappa.\tau_1' \,|\, \mathcal{L}_1$ and $\Delta \vdash \tau_2 \,|\, \mathcal{L}_2$, and by inversion $\Delta, \alpha{:}\kappa \vdash \tau_1 \,|\, \mathcal{L}_1$. Then we can apply Lemma C.15 and we are done.

□

Lemma C.17 (Well formed terms possess well formed types). *If* $\Delta; \Gamma \vdash e : \tau \,|\, \Sigma$ *then* $\Delta \vdash \tau : \star$.

Proof. Straightforward induction on typing derivations. □

Lemma C.18 (Reduction respects type equivalence). *The following hold:*

- *If* $\Delta \vdash \tau_1 \Downarrow \tau_1'$ *and* $\Delta \vdash \tau_1 : \kappa$ *then* $\Delta \vdash \tau_1 = \tau_1' : \kappa$.
- *If* $\Delta \vdash \tau_1 \Downarrow_* \tau_1'$ *and* $\Delta \vdash \tau_1 : \kappa$ *then* $\Delta \vdash \tau_1 = \tau_1' : \kappa$.
- *If* $\Delta \vdash \tau_1 \downarrow \tau_1'$ *and* $\Delta \vdash \tau_1 : \kappa$ *then* $\Delta \vdash \tau_1 = \tau_1' : \kappa$.

Proof. The first case is easy induction on the weak head reduction relation. The second case can be proved by induction on the number of steps of reduction, appealing to the first case. The third case follows by a simple case analysis on the weak head normalization derivation and appealing to the second case. □

Lemma C.19 (Plug-in property for type equivalence). *If* $\Delta \vdash \tau_1 \downarrow \tau_1'$ *and* $\Delta \vdash \tau_1\ \tau_2 : \star$ *then* $\Delta \vdash \tau_1\ \tau_2 = \tau_1'\ \tau_2 : \star$.

Proof. The result follows easily from Lemma C.18 and an application of the teq-core:app-con rule. □

Lemma C.20 (Weakening for signatures). *If* $\Delta; \Gamma \vdash e : \tau \,|\, \Sigma$ *and* $\Sigma \subseteq \Sigma'$ *then* $\Delta; \Gamma \vdash e : \tau \,|\, \Sigma'$.

Proof. Easy induction on typing derivations. □

Corollary C.21 (Type normalization preserves kinds). *If* $\Delta \vdash \tau \downarrow \tau'$ *and* $\Delta \vdash \tau : \kappa$ *then* $\Delta \vdash \tau' : \kappa$.

Proof. By induction on the weak head normalization relation. We proceed by case analysis on the kind κ. If $\kappa = \star$, then observe that the rule applied was whn-core:star and the result follows from Lemma C.14. Suppose now that $\kappa = \kappa_1 \to \kappa_2$. Then the rule applied was whn-core:simple-con and then it must be that $\Delta, \alpha{:}\kappa_1 \vdash \tau\alpha : \kappa_2$ and $\Delta, \alpha{:}\kappa_1 \vdash \tau\alpha \downarrow \tau_1$. But then by induction hypothesis we get that $\Delta, \alpha{:}\kappa_1 \vdash \tau_1 : \kappa_2$ and we can apply the twf-core:abs to get $\Delta \vdash \lambda\alpha{:}\kappa_1.\tau_1 : \kappa_1 \to \kappa_2$. □

Corollary C.22 (Type normalization preserves label set analysis). *If* $\Delta \vdash \tau \downarrow \tau'$, $\Delta \vdash \tau : \kappa$ *and* $\Delta \vdash \tau \,|\, \mathcal{L}$ *then* $\Delta \vdash \tau' \,|\, \mathcal{L}'$, *with* $\Delta \vdash \mathcal{L}' \sqsubseteq \mathcal{L}$.

Proof. By induction on weak head normalization relation. We proceed once again with case analysis on the kind κ. If $\kappa = \star$ then the result follows from Lemma C.16. Suppose now that $\kappa = \kappa_1 \to \kappa_2$. Then it must be that $\tau' = \lambda\alpha{:}\kappa_1.\tau_2$ and $\Delta, \alpha{:}\kappa_1 \vdash \tau\alpha \downarrow \tau_2$. Since $\Delta{:}\alpha{:}\kappa_1 \vdash \alpha \,|\, \varnothing$ and by weakening $\Delta{:}\alpha{:}\kappa_1 \vdash \tau \,|\, \mathcal{L}$ we get that $\Delta{:}\alpha{:}\kappa_1 \vdash \tau\alpha \,|\, \varnothing \cup \mathcal{L}$ and by induction hypothesis $\Delta, \alpha{:}\kappa_1 \vdash \tau_2 \,|\, \mathcal{L}_2$ with $\Delta, \alpha{:}\kappa_1 \vdash \mathcal{L}_2 \sqsubseteq \varnothing \cup \mathcal{L}$. Then by tan-core:abs we get $\Delta \vdash \lambda\alpha{:}\kappa_1.\tau_2 \,|\, \mathcal{L}_2$ and by weakening for the label set subset relation $\Delta \vdash \mathcal{L}_2 \sqsubseteq \mathcal{L}$. □

Theorem C.23 (Applications of expressions with polykinded types). *If the following conditions hold:*

- $\Delta \vdash \tau \downarrow \sigma\tau_1 \ldots \tau_n = \rho_n[\sigma]$
- $\Delta \vdash \tau : \star$
- $\Delta \vdash \tau_i : \kappa_i$, $\Delta \vdash \tau_i \,|\, \mathcal{L}_i$, $\Delta \vdash \mathcal{L}_i \sqsubseteq \mathcal{L}$, *for all* $1 \leq i \leq n$.
- $\Delta; \Gamma \vdash e : \tau' \langle \sigma : \kappa^{(n)} \restriction \mathcal{L} \rangle \,|\, \Sigma$, *where* $\kappa^{(n)} = \kappa_1 \to (\kappa_2 \to \ldots (\kappa_n \to \star) \ldots)$.

Then $\Delta; \Gamma \vdash p_n[e] : \tau'\tau \,|\, \Sigma$, *where* $\rho \rightsquigarrow p$.

Proof. The proof is by induction on n. For $n = 0$ we have that $\Delta \vdash \tau \downarrow \sigma$ and by Corollary C.21 $\Delta \vdash \sigma : \star$. Then we have that $\Delta; \Gamma \vdash e : \tau' \langle \sigma : \star \restriction \mathcal{L} \rangle \,|\, \Sigma$, which by definition is $\Delta; \Gamma \vdash e : \tau'\sigma \,|\, \Sigma$. Now, by Lemma C.18 we get that $\Delta \vdash \tau = \sigma : \star$, hence $\Delta; \Gamma \vdash e : \tau'\tau \,|\, \Sigma$ or $\Delta; \Gamma \vdash p_0[e] : \tau'\tau \,|\, \Sigma$. Suppose now that the property holds for any paths of order less or equal to n and take $\Delta \vdash \tau \downarrow \rho_{n+1}[\sigma] = (\sigma\tau_1) \ldots \tau_n\tau_{n+1} = \rho'[(\sigma\tau_1)]$. Then, by Corollary C.21 we have that $\Delta \vdash \rho_{n+1}[\sigma] : \star$ and by repeated inversions it must be that $\Delta \vdash \sigma : \kappa^{(n+1)} = \kappa_1 \to (\ldots (\kappa_{n+1} \to \star) \ldots)$. Now we know that $\Delta; \Gamma \vdash e : \tau' \langle \sigma : \kappa^{(n+1)} \restriction \mathcal{L} \rangle \,|\, \Sigma$ which means that $\Delta; \Gamma \vdash e : \tau' \langle \sigma : \kappa_1 \to (\ldots (\kappa_{n+1} \to \star) \ldots) \restriction \mathcal{L} \rangle \,|\, \Sigma$. But this is equivalent to $\Delta; \Gamma \vdash e : \forall \alpha{:}\kappa_1 \restriction \mathcal{L}.\tau' \langle \sigma\alpha : \kappa_2 \to (\ldots (\kappa_{n+1} \to \star) \ldots) \restriction \mathcal{L} \rangle \,|\, \Sigma$ and this means that $\Delta; \Gamma \vdash e\tau_1 : \tau' \langle \sigma\tau_1 : \kappa_2 \to (\ldots (\kappa_{n+1} \to \star) \ldots) \restriction \mathcal{L} \rangle \,|\, \Sigma$. But now by induction hypothesis $\Delta; \Gamma \vdash p'_n[e\tau_1] : \tau'\tau \,|\, \Sigma$, but note that $p_{n+1}[e] = p'_n[e\tau_1]$, therefore $\Delta; \Gamma \vdash p_{n+1}[e] : \tau'\tau \,|\, \Sigma$. □

In the following we use the abbreviation h_f to refer to one of the following forms of types: $\forall\phi$, $\mathcal{L}_1 \Rightarrow \tau \restriction \mathcal{L}_2$, $\rho[\alpha]$, $\rho[l]$.

Lemma C.24. *If* $\Delta \vdash \tau : \kappa$ *and* $\Delta \vdash \tau \Downarrow_* \tau'$, $\Delta \vdash \tau' \not\Downarrow$, *then* $\tau' = h_f$ *or* $\tau' = \lambda\alpha{:}\kappa.\tau_1$, *where* $\Delta, \alpha{:}\kappa \vdash \tau_1 \not\Downarrow$.

Proof. Since by Lemma C.14, $\Delta \vdash \tau' : \kappa$, suppose by contradiction that τ' is not one of the above forms. Clearly there are two cases:

- $\tau' = \lambda\alpha{:}\kappa.\tau_1$ with $\Delta, \alpha{:}\kappa \vdash \tau_1 \Downarrow \tau'_1$, but then we can apply whr-core:abs-con and derive a contradiction to the fact that $\Delta \vdash \tau' \not\Downarrow$.
- $\tau' = \rho[\tau_1]$, where $\tau_1 \neq \alpha$, $\tau_1 \neq l$ and τ_1 is applied to at least one argument. Suppose $\tau' = \tau_1 \ldots \tau_n$, $n \geq 2$. We will prove by induction on n that $\Delta \vdash \tau' \Downarrow \tau''$, which will be a contradiction. Suppose $n = 2$ and then $\tau' = \tau_1\tau_2$. We know that $\Delta \vdash \tau' : \kappa$ and by inversion it must be that $\Delta \vdash \tau_1 : \kappa' \to \kappa$. But τ_1 is not an application, therefore it must be that $\tau_1 = \lambda\alpha{:}\kappa'.\tau''$, but then we can apply whr-core:abs-beta and then $\Delta \vdash \tau' \Downarrow \tau''[\tau_2/\alpha]$. Suppose now that $n \rangle 2$, and that $\Delta \vdash \tau_1 \ldots \tau_n \Downarrow \tau''$. Then by whr-core:app-con we have that $\Delta \vdash \tau_1 \ldots \tau_n\tau_{n+1} \Downarrow \tau''\tau_{n+1}$. So, in all cases $\Delta \vdash \tau' \Downarrow \tau''$ for some τ'', a contradiction. So τ' has the desired form.

□

Corollary C.25. *If* $\vdash \tau : \star$ *and* $\vdash \tau \,|\, \mathcal{L}$, *then* $\vdash \tau \downarrow \rho[l]$ *for some* l, *such that* $\vdash \{l\} \sqsubseteq \mathcal{L}$.

Proof. Because $\vdash \tau : \star$, then only possibility for normalization, is the rule whn-core:star, that is $\vdash \tau \Downarrow_* \tau'$ for some τ' and $\vdash \tau' \not\Downarrow$. But by Lemma C.24, τ has one of the forms h_f; it cannot be an abstraction because $\vdash \tau : \star$. Since $\vdash \tau \,|\, \mathcal{L}$, by Lemma C.16 we get that $\vdash \tau' \,|\, \mathcal{L}'$ for some $\mathcal{L}'$, such that $\vdash \mathcal{L}' \sqsubseteq \mathcal{L}$. But then τ cannot be a universal type, neither a map type because these types do not belong in the label set analysis relation. Moreover τ' doesn't contain free variables therefore it cannot be a $\rho[\alpha]$. So it must be $\rho[l]$. But by the label set analysis for application, since we have that $\vdash \rho[l] \,|\, \mathcal{L}'$ it must be that $\vdash \{l\} \sqsubseteq \mathcal{L}'$, and since $\vdash \mathcal{L}' \sqsubseteq \mathcal{L}$ we have that $\vdash \{l\} \sqsubseteq \mathcal{L}$. □

Theorem C.26 (Type Weak Head Forms). *If* $\Delta \vdash \tau_1 : \kappa$ *and* $\Delta \vdash \tau_1 \downarrow \tau'_1$, *then* $\tau'_1 = \lambda\overline{\alpha}{:}\overline{\kappa}.h_f$ *where the syntax* $\lambda\overline{\alpha}{:}\overline{\kappa}.h_f$ *is an abbreviation for* $\lambda\alpha_1{:}\kappa_1 \ldots . \lambda\alpha_n{:}\kappa_n.h_f$ *and for* $n = 0$ *the expression is just a head form* h_f.

Proof. The proof is by induction on the structure of κ. For $\kappa = \star$, the result follows by a specialization of Lemma C.24 for kind $\star$. Suppose that $\kappa = \kappa_1 \to \kappa_2$. Then clearly it must be that $\Delta, \alpha_1{:}\kappa_1 \vdash \tau_1\alpha \downarrow \tau''$ and $\tau' = \lambda\alpha_1{:}\kappa_1.\tau''$. But then $\Delta, \alpha_1{:}\kappa_1 \vdash \tau_1\alpha : \kappa_2$, and by induction hypothesis $\tau'' = \lambda\alpha_2{:}\kappa_2.\ldots.\lambda\alpha_n{:}\kappa_n.h_f$ and then $\tau' = \lambda\overline{\alpha}{:}\overline{\kappa}.h_f$. □

In the following we use the notation $\tau_1 \sim \tau_2$ for types that have the same head form. The next two lemmas establish the fact that equivalent types reduce to types that have the same head forms, that is, they are both paths of labels, or paths of variables, or map types or the same kind of universal types.

Lemma C.27. *If $\Delta \vdash (\lambda\alpha{:}\kappa'.\sigma)\tau_1\overline{\tau} : \kappa$ then if $\Delta \vdash (\lambda\alpha{:}\kappa'.\sigma)\tau_1\overline{\tau} \downarrow \tau_1'$ and $\Delta \vdash \sigma[\tau_1/\alpha]\overline{\tau} \downarrow \tau_2'$, then $\tau_1' = \tau_2'$.*

Proof. The proof is by structural induction on the kind κ. For kind $\star$ it must be that $\Delta \vdash (\lambda\alpha_1{:}\kappa_1.\sigma)\tau_1\overline{\tau} \Downarrow_* \tau_1'$, but we know that $\Delta \vdash (\lambda\alpha_1{:}\kappa_1.\sigma)\tau_1\overline{\tau} \Downarrow \sigma[\tau_1/\alpha_1]\overline{\tau}$, which means that the two types normalize to exactly the same type eventually. Suppose now that $\kappa = \kappa_1 \to \kappa_2$. Then it must be the case that $\Delta{:}\alpha_1{:}\kappa_1 \vdash (\lambda\alpha{:}\kappa'.\sigma)\tau_1\overline{\tau}\alpha_1 \downarrow \tau_1''$ and $\tau_1' = \lambda\alpha_1{:}\kappa_1.\tau_1''$. Then, observe that $\Delta \vdash \sigma[\tau_1/\alpha]\overline{\tau}\alpha_1 : \kappa_2$ and suppose $\Delta, \alpha_1{:}\kappa_1 \vdash \sigma[\tau_1/\alpha]\overline{\tau}\alpha_1 \downarrow \tau_2''$. Then also $\tau_2' = \lambda\alpha_1{:}\kappa_1.\tau_2''$. Since κ_2 is structurally smaller than κ, by induction hypothesis $\tau_1'' = \tau_2''$, which implies $\tau_1' = \tau_2'$. □

Theorem C.28 (Weak head normalization respects type equivalence). *If $\Delta \vdash \tau_1 = \tau_2 : \kappa$, $\Delta \vdash \tau_1 \downarrow \tau_1'$, $\Delta \vdash \tau_2 \downarrow \tau_2'$, then $\tau_1' = \lambda\overline{\alpha}{:}\overline{\kappa}.h_{f_1}$, and $\tau_2' = \lambda\overline{\alpha}{:}\overline{\kappa}.h_{f_2}$, with $h_{f_1} \sim h_{f_2}$.*

Proof. By induction on the type equivalence derivation $\Delta \vdash \tau_1 = \tau_2 : \kappa$. The cases for teq-core:refl, teq-core:sym, teq-core:trans, teq-core:map-con, teq-core:tall-con, teq-core:lall-con, teq-con:sall-con are easy. Suppose that we are considering the case of teq-core:abs-beta. We have that $\Delta \vdash (\lambda\alpha{:}\kappa_1.\tau_1)\tau_2 = \tau_1[\tau_2/\alpha] : \kappa_2$, given that $\Delta \vdash \lambda\alpha{:}\kappa_1.\tau_1 : \kappa_1 \to \kappa_2$ and $\Delta \vdash \tau_2 : \kappa_1$. The result follows from application of Lemma C.27. The rest of the cases are easy to verify and rely on simple applications of the inductive hypothesis. □

C.4 Canonical Forms

Now we are ready to present the canoncal forms theorem.

Theorem C.29 (Canonical Forms). *The following hold:*

1. *If $; \vdash v : \ell_{\mathsf{int}} \,|\, \Sigma$, then $v = i$.*
2. *If $; \vdash v : \sigma_1 \to \sigma_2 \,|\, \Sigma$, then $v = \lambda x{:}\tau_1.e$.*
3. *If $; \vdash v : p[l] \,|\, \Sigma$, then $v = \{\!\{v'\}\!\}^+_{l=\tau}$, if $l \neq \ell_{\mathsf{int}}, \ell_\to$.*
4. *If $; \vdash v : \mathcal{L}_1 \Rightarrow \tau \restriction \mathcal{L}_2 | \,\Sigma$, then $v = \varnothing$ or $v = \{l \Rightarrow e\}$, or $v = v_1 \bowtie v_2$.*
5. *If $; \vdash v : \forall\alpha{:}\kappa \restriction \mathcal{L}.\tau \,|\, \Sigma$, then $v = \Lambda\alpha{:}\kappa \restriction \mathcal{L}'.e$.*
6. *If $; \vdash v : \forall\iota{:}\mathrm{L}(\kappa).\tau \,|\, \Sigma$, then $v = \Lambda\iota{:}\mathrm{L}(\kappa).e$.*
7. *If $; \vdash v : \forall s{:}\mathrm{Ls}.\tau \,|\, \Sigma$, then $v = \Lambda s{:}\mathrm{Ls}.e$.*

Proof. The proof is by contradiction appealing to Lemma C.28. We only show one case, the others are similar. Suppose we are examining the first case, and by contradiction assume that $v \neq i$. Suppose for example that it is $\lambda x{:}\tau_1.e$. Then since it is well typed there is a typing derivation that uses wf-core:abs and the equivalence rule, giving a type of the form $\tau_1 \to \tau_2$ to this abstraction. Then it must be eventually that $\vdash \ell_{\mathsf{int}} = \tau_1 \to \tau_2 : \star$. But now by Lemma C.28, $\vdash \ell_{\mathsf{int}} \downarrow h_{f_1}$ and $\vdash \tau_1 \to \tau_2 \downarrow h_{f_2}$, and $h_{f_1} \sim h_{f_2}$. But ℓ_{int} cannot be reduced further nor can the function type, but they are different head forms, clearly a contradiction to the fact that $h_{f_1} \sim h_{f_2}$. The rest of the cases use similar arguments. □

Corollary C.30 (Canonical Forms with Inversion). *We provide some more information building on the cases of the previous canonical forms theorem.*

- *Case 2 also implies that* $\vdash \tau_1 = \sigma_1 : \star$.
- *Case 3 also implies that* $l{:}\kappa = \tau \in \Sigma$.
- *Case 4 also implies that if* $v = \varnothing$ *then* $\mathcal{L}_1 = \varnothing$, *if* $v = \{l \Rightarrow e\}$ *then* $\vdash \{l\} \sqsubseteq \mathcal{L}$, *if* $v = v_1 \bowtie v_2$ *then* $; \vdash v_1 : \mathcal{L}_{11} \Rightarrow \tau' \restriction \mathcal{L}_{21} | \Sigma$, $; \vdash v_1 : \mathcal{L}_{21} \Rightarrow \tau' \restriction \mathcal{L}_{22} | \Sigma$ *and* $\vdash \mathcal{L}_1 = \mathcal{L}_{11} \cup \mathcal{L}_{21}$.
- *Case 5 also implies* $\vdash \mathcal{L} = \mathcal{L}'$.

Proof. We just combine the results of the canonical forms lemma and the inversion on the typing relation. □

Corollary C.31 (Canonical map values). *If* $; \vdash v : \mathcal{L}_1 \Rightarrow \tau \restriction \mathcal{L}_2 | \Sigma$ *and* $\vdash \{l\} \sqsubseteq \mathcal{L}_1$ *then* $v = v_1 \bowtie \ldots \bowtie v_n$, *for* $n \geq 0$ *and* $\exists\, v_i$, *such that* $v_i = \{l \Rightarrow e\}$.

Proof. By induction on the size of v. By the canonical forms lemma we have three cases for v.

- $v = \varnothing$. Then $\mathcal{L}_1 = \varnothing$ and the implication trivially holds since the assumptions are not satisfied.
- $v = \{l_1 \Rightarrow e\}$. Then it must be that $\mathcal{L}_1 = \{l_1\}$ and since $\vdash \{l\} \sqsubseteq \mathcal{L}_1$ we have that $l_1 = l$.
- $v = v_1 \bowtie v_2$. The result follows from the inversion and application of the inductive hypothesis, for v_1 or v_2, depending on whether l is contained in the label set of v_1 or the label set of v_2.

□

C.5 Substitution Lemmas

C.5.1 Label Substitutions

Lemma C.32. *If* $\Delta, \iota{:}\mathrm{L}(\kappa) \vdash \sigma : \kappa_2$ *and* $\Delta \vdash l : \mathrm{L}(\kappa)$, *then* $\Delta[l/\iota] \vdash \iota[\sigma/l] : \kappa_2$.

Proof. Straightforward induction. □

Lemma C.33. *If* $\Delta, \iota{:}\mathrm{L}(\kappa) \vdash \sigma_1 = \sigma_2 : \kappa_2$ *and* $\Delta \vdash l : \mathrm{L}(\kappa)$, *then* $\Delta[l/\iota] \vdash \sigma_1[l/\iota] = \sigma_2[l/\iota] : \kappa_2$.

Proof. Straightforward induction. □

Lemma C.34. *If* $\Delta, \iota{:}\mathrm{L}(\kappa) \vdash \tau \mid \mathcal{L}$ *and* $\Delta \vdash l : \mathrm{L}(\kappa)$, *then* $\Delta[l/\iota] \vdash \tau[l/\iota] \mid \mathcal{L}[l/\iota]$.

Proof. Straightforward induction. □

Lemma C.35. *If* $\Delta, \iota{:}\mathrm{L}(\kappa) \vdash \mathcal{L}_1 \sqsubseteq \mathcal{L}_2$ *and* $\Delta \vdash l : \mathrm{L}(\kappa)$, *then* $\Delta[l/\iota] \vdash \mathcal{L}_1[l/\iota] \sqsubseteq \mathcal{L}_2[l/\iota]$.

Proof. Straightforward induction. □

Lemma C.36. *If* $\Delta, \iota{:}\mathrm{L}(\kappa) \vdash l_1 : \mathrm{L}(\kappa_1)$ *and* $\Delta \vdash l : \mathrm{L}(\kappa)$, *then* $\Delta[l/\iota] \vdash l_1[l/\iota] : \mathrm{L}(\kappa_1)$.

Proof. Straightforward induction. □

Lemma C.37. *If* $\Delta, \iota{:}\mathrm{L}(\kappa) \vdash \mathcal{L} : \textsc{Ls}$ *and* $\Delta \vdash l : \mathrm{L}(\kappa)$, *then* $\Delta[l/\iota] \vdash \mathcal{L}[l/\iota] : \textsc{Ls}$.

Proof. Straightforward induction. □

Theorem C.38 (Label Substitution Lemma). *If* $\Delta, \iota{:}\mathrm{L}(\kappa); \Gamma \vdash e : \sigma \mid \Sigma$ *and* $\Delta \vdash l : \mathrm{L}(\kappa)$, *then* $\Delta[l/\iota]; \Gamma[l/\iota] \vdash e[l/\iota] : \sigma[l/\iota] \mid \Sigma[l/\iota]$.

Proof. Straightforward induction on typing derivations using the above lemmas. □

C.5.2 Label Set Substitutions

Lemma C.39. *If* $\Delta, s{:}\text{Ls} \vdash \sigma : \kappa$ *and* $\Delta \vdash \mathcal{L} : \text{Ls}$*, then* $\Delta[\mathcal{L}/s] \vdash \sigma[\mathcal{L}/s] : \kappa$*.*

Proof. Straightforward induction. □

Lemma C.40. *If* $\Delta, s{:}\text{Ls} \vdash \sigma_1 = \sigma_2 : \kappa$ *and* $\Delta \vdash \mathcal{L} : \text{Ls}$*, then* $\Delta[\mathcal{L}/s] \vdash \sigma_1[\mathcal{L}/s] = \sigma_2[\mathcal{L}/s] : \kappa$*.*

Proof. Straightforward induction. □

Lemma C.41. *If* $\Delta, s{:}\text{Ls} \vdash \sigma \mid \mathcal{L}_1$ *and* $\Delta \vdash \mathcal{L} : \text{Ls}$*, then* $\Delta[\mathcal{L}/s] \vdash \sigma[\mathcal{L}/s] \mid \mathcal{L}_1[\mathcal{L}/s]$*.*

Proof. Straightforward induction. □

Lemma C.42. *If* $\Delta, s{:}\text{Ls} \vdash \mathcal{L}_1 \sqsubseteq \mathcal{L}_2$ *and* $\Delta \vdash \mathcal{L} : \text{Ls}$*, then* $\Delta[\mathcal{L}/s] \vdash \mathcal{L}_1[\mathcal{L}/s] \sqsubseteq \mathcal{L}_2[\mathcal{L}/s]$*.*

Proof. Straightforward induction. □

Lemma C.43. *If* $\Delta, s{:}\text{Ls} \vdash \mathcal{L}_1 : \text{Ls}$ *and* $\Delta \vdash \mathcal{L} : \text{Ls}$*, then* $\Delta[\mathcal{L}/s] \vdash \mathcal{L}_1[\mathcal{L}/s] : \text{Ls}$*.*

Proof. Straightforward induction. □

Theorem C.44 (Label Set Substitution Lemma). *If* $\Delta, s{:}\text{Ls}; \Gamma \vdash e : \sigma \mid \Sigma$ *and* $\Delta \vdash \mathcal{L} : \text{Ls}$*, then* $\Delta[\mathcal{L}/s]; \Gamma[\mathcal{L}/s] \vdash e[\mathcal{L}/s] : \sigma[\mathcal{L}/s] \mid \Sigma[\mathcal{L}/s]$*.*

Proof. Straightforward induction on typing derivations using the above lemmas. □

C.5.3 Type Substitutions

Lemma C.45. *If* $\Delta, \alpha{:}\kappa \restriction \mathcal{L} \vdash \sigma : \kappa_1$ *and* $\Delta \vdash \tau : \kappa$*, then* $\Delta \vdash \sigma[\tau/\alpha] : \kappa_1$*.*

Proof. Straightforward induction on typing derivations. □

Lemma C.46. *If* $\Delta, \alpha{:}\kappa \restriction \mathcal{L} \vdash \sigma_1 = \sigma_2 : \kappa_2$ *and* $\Delta \vdash \tau : \kappa$*, then* $\Delta \vdash \sigma_1[\tau/\alpha] = \sigma_2[\tau/\alpha] : \kappa_2$*.*

Proof. Straightforward induction on the type equivalence derivations. □

Lemma C.47. *If* $\Delta, \alpha{:}\kappa \restriction \mathcal{L} \vdash \tau_1 \mid \mathcal{L}_1$ *and* $\Delta \vdash \tau_2 : \kappa$*,* $\Delta \vdash \tau_2 \mid \mathcal{L}_0$*,* $\Delta \vdash \mathcal{L}_0 \sqsubseteq \mathcal{L}_1$*, then* $\Delta \vdash \tau_1[\tau_2/\alpha] \mid \mathcal{L}'$*, with* $\Delta \vdash \mathcal{L}' \sqsubseteq \mathcal{L}_0 \cup \mathcal{L}_1$*.*

Proof. By induction on the label set analysis derivations. The proof is similar to that of Lemma C.15 and therefore ommited. □

Theorem C.48. *If* $\Delta, \alpha{:}\kappa \restriction \mathcal{L}; \Gamma \vdash e : \sigma \mid \Sigma$ *and* $\Delta \vdash \tau : \kappa$ *and* $\Delta \vdash \tau \mid \mathcal{L}_1$ *and* $\Delta \vdash \mathcal{L}_1 \sqsubseteq \mathcal{L}$*, then* $\Delta; \Gamma[\tau/\alpha] \vdash e[\tau/\alpha] : \sigma[\tau/\alpha] \mid \Sigma[\tau/\alpha]$*.*

Proof. The proof is by induction on the typing derivations appealing to the lemmas above. The only interesting cases are those of wf-core:typecase and wf-core:tapp.

- Case wf-core:typecase. Here we have that $\Delta, \alpha{:}\kappa \restriction \mathcal{L}; \Gamma \vdash \textbf{typecase}\ \tau_0\ e : \tau'\tau \mid \Sigma$, given that
  $\Delta, \alpha{:}\kappa \restriction \mathcal{L}; \Gamma \vdash e : \mathcal{L}_{a1} \Rightarrow \tau' \restriction \mathcal{L}_{a2} \mid \Sigma$, $\Delta, \alpha{:}\kappa \restriction \mathcal{L} \vdash \tau_0 : \star$, $\Delta, \alpha{:}\kappa \restriction \mathcal{L} \vdash \tau_0 \mid \mathcal{L}_0$, $\Delta, \alpha{:}\kappa \restriction \mathcal{L} \vdash \mathcal{L}_{a1} \sqsubseteq \mathcal{L}_{a2}$ and $\Delta, \alpha{:}\kappa \restriction \mathcal{L} \vdash \mathcal{L}_0 \sqsubseteq \mathcal{L}_{a1}$. Now, by Lemma C.47 we get that $\Delta \vdash \tau_0[\tau/\alpha] \mid \mathcal{L}'$ with $\Delta \vdash \mathcal{L}' \sqsubseteq \mathcal{L}_0 \cup \mathcal{L}_1$. Also, by Lemma C.45 $\Delta \vdash \tau_0[\tau/\alpha] : \star$. Also label sets do not contain type variables, so $\Delta \vdash \mathcal{L}_{a1} \sqsubseteq \mathcal{L}_{a2}$ and $\Delta \vdash \mathcal{L}_0 \sqsubseteq \mathcal{L}_{a1}$. Since $\Delta \vdash \mathcal{L}' \sqsubseteq \mathcal{L}_0 \cup \mathcal{L}_1$ it must be that $\Delta \vdash \mathcal{L}' \sqsubseteq \mathcal{L}_0$ and $\Delta \vdash \mathcal{L}' \sqsubseteq \mathcal{L}_1$. Also, by induction hypothesis we have that $\Delta; \Gamma[\tau/\alpha] \vdash e[\tau/\alpha] : \mathcal{L}_{a1} \Rightarrow \tau'[\tau/\alpha] \restriction \mathcal{L}_{a2} \mid \Sigma[\tau/\alpha]$, and now we have the premises of wf-core:typecase and we can apply the rule to get the result.

- Case wf-core:tapp. In this case $\Delta, \alpha{:}\kappa \restriction \mathcal{L}; \Gamma \vdash e[\tau_1] : \sigma[\tau_1/\alpha] \mid \Sigma$, given that

  $\Delta, \alpha{:}\kappa \restriction \mathcal{L}; \Gamma \vdash e : \forall \beta{:}\kappa_1 \restriction \mathcal{L}_2.\sigma \mid \Sigma$, $\Delta, \alpha{:}\kappa \restriction \mathcal{L} \vdash \tau_1 : \kappa_1$, $\Delta, \alpha{:}\kappa \restriction \mathcal{L} \vdash \tau_1 \mid \mathcal{L}'$, $\Delta, \alpha{:}\kappa \restriction \mathcal{L} \vdash \mathcal{L}' \sqsubseteq \mathcal{L}_2$. By induction hypothesis we get that $\Delta; \Gamma[\tau/\alpha] \vdash e[\tau/\alpha] : \forall \beta{:}\kappa_1 \restriction \mathcal{L}_2.\sigma[\tau/\alpha] \mid \Sigma[\tau/\alpha]$. By Lemma C.45 we get that $\Delta \vdash \tau_1[\tau/\alpha] : \kappa_1$ and by Lemma C.47 we get $\Delta \vdash \tau_1[\tau/\alpha] \mid \mathcal{L}''$, with $\Delta \vdash \mathcal{L}'' \sqsubseteq \mathcal{L}_1 \cup \mathcal{L}'$, which implies $\Delta \vdash \mathcal{L}'' \sqsubseteq \mathcal{L}'$. Also, because label sets do not contain type variable we get $\Delta \vdash \mathcal{L}' \sqsubseteq \mathcal{L}_2$ and by transitivity we $\Delta \vdash \mathcal{L}'' \sqsubseteq \mathcal{L}_2$. But now we have all the requirements of wf-core:tapp rule and by applying it we are done.

□

C.5.4 Term Substitutions

Theorem C.49. *If* $\Delta; \Gamma, x{:}\sigma_1 \vdash e_1 : \sigma_2 \mid \Sigma$ *and* $\Delta; \Gamma \vdash e_2 : \sigma_1 \mid \Sigma$*, then* $\Delta; \Gamma \vdash e_1[e_2/x] : \sigma_2 \mid \Sigma$.

Proof. Easy induction on typing derivations. □

C.6 Progress

Theorem C.50. *If* $; \vdash e : \tau \mid \Sigma$ *and* $\ell_{\mathsf{int}}, \ell_{\rightarrow} \notin dom(\Sigma)$*, then either* $e = v$*, or if* $\mathcal{L} = dom(\Sigma) \cup \{\ell_{\mathsf{int}}\} \cup \{\ell_{\rightarrow}\}$*, then* $\mathcal{L}; e \mapsto \mathcal{L}'; e'$.

Proof. The proof is by induction on typing derivations. The proof is a standard progress proof. We only show the interesting cases; we only do a proof sketch for standard easy cases.

- Case wf-core:typecase. We have that $; \vdash \mathbf{typecase}\ \tau\ e : \tau'\tau \mid \Sigma$, given that $; \vdash e : \mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L}_2 \mid \Sigma$, $\vdash \tau : \star$, $\vdash \tau \mid \mathcal{L}$, $\vdash \mathcal{L}_1 \sqsubseteq \mathcal{L}_2$, $\vdash \mathcal{L} \sqsubseteq \mathcal{L}_1$. By induction hypothesis, either $\mathcal{L}; e \mapsto \mathcal{L}'; e'$, or $e = v$. In the first case we just apply ev-core:typecase-con and in the second case, by Corollary C.25 we get that $\vdash \tau \downarrow \rho[l]$, such that $\vdash \{l\} \sqsubseteq \mathcal{L}$ and by transitivity we get $\vdash \{l\} \sqsubseteq \mathcal{L}_1$. Then, by Corollary C.31 we get that $v = v_1 \bowtie \ldots \bowtie v_n$ where $\exists i$, such that $v_i = \{l \Rightarrow e\}$ and then we can apply ev-core:typecase.

- Case wf-core:hin. Here we have $; \vdash \{\!\{e : \tau'\}\!\}^{+}_{l=\tau} : \tau'\ l \mid \Sigma$, given that $; \vdash e : \tau'\ \tau \mid \Sigma$ and $l{:}\kappa = \tau \in \Sigma$. By induction hypothesis, either $e = v$ or $\mathcal{L}; e \mapsto \mathcal{L}'; e'$. In the second case we can just apply ev-core:hcolor-con and we are done. Suppose now that $e = v$, so actually our assumptions are that $; \vdash \{\!\{v : \tau'\}\!\}^{+}_{l=\tau} : \tau'\ l \mid \Sigma$, given that $; \vdash v : \tau'\ \tau \mid \Sigma$ and $l{:}\kappa = \tau \in \Sigma$. By Lemma C.17 we have that $\vdash \tau'\tau : \star$, so by inversion we see that $\vdash \tau' : \kappa \rightarrow \star$. By a simple corollary of Lemma C.26 we have the following cases for the normalization of τ:.

  - $\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\rho[\alpha]$. In this case we can just apply ev-core:hcolor-base.
  - $\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\ell_{\mathsf{int}}$. Now we know by Lemma C.17 that $\vdash \tau'\tau : \star$. By Corollary C.19 we get that $\vdash \tau'\tau = (\lambda\alpha{:}\kappa.\ell_{\mathsf{int}})\tau : \star$, which using teq-core:abs-beta and teq-core:trans gives $\vdash \tau'\tau = \ell_{\mathsf{int}} : \star$. Finally, by wf-core:weak we get that $; \vdash v : \ell_{\mathsf{int}} \mid \Sigma$. But now by the canonical forms lemma $v = i$ and we can apply ev-core:hcolor-int to take a step.
  - $\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\tau_1 \rightarrow \tau_2$. Again by Lemma C.17 and Corollary C.19 we get that $\vdash \tau'\tau = (\lambda\alpha{:}\kappa.\tau_1 \rightarrow \tau_2)\tau : \star$, which using teq-core:abs-beta and teq-core:trans yields $\vdash \tau'\tau = \tau_1[\tau/\alpha] \rightarrow \tau_2[\tau/\alpha] : \star$. Finally, by wf-core:weak we get that $; \vdash v : \tau_1[\tau/\alpha] \rightarrow \tau_2[\tau/\alpha] \mid \Sigma$, which by the canonical forms lemma gives us that $v = \lambda x{:}\sigma_1.e$ with (using inversion) $\vdash \sigma_1 = \tau_1[\tau/\alpha] : \star$ and now we can apply ev-core:color-abs1 to take a step.
  - $\vdash \tau \downarrow \lambda\alpha{:}\kappa.\rho[l_1]$ where $l_1 \neq \ell_{\mathsf{int}}, \ell_{\rightarrow}$. Once more, by Lemma C.17 and Corollary C.19 we get that $\vdash \tau'\tau = (\lambda\alpha{:}\kappa.\rho[l_1])\tau : \star$ and then by teq-core:abs-beta and teq-core:trans we get that $\vdash \tau'\tau = \rho[\tau/\alpha][l_1] : \star$ or $\vdash \tau'\tau = \rho'[l_1] : \star$ where $\rho' = \rho[\tau/\alpha]$. Finally, by wf-core:weak we get that $; \vdash v : \rho'[l_1] \mid \Sigma$ and by canonical forms lemma it must be that $v = \{\!\{v'\}\!\}^{+}_{l_1=\tau_1}$ and we can apply ev-core:hcolor-color rule to take a step.

- $\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\mathcal{L}_1 \Rightarrow \tau_1 \restriction \mathcal{L}_2$. By Lemma C.17 and Corollary C.19 we get that $\vdash \tau'\tau = (\lambda\alpha{:}\kappa.\mathcal{L}_1 \Rightarrow \tau_1 \restriction \mathcal{L}_2)\tau : \star$ and by teq-core:abs-beta and teq-core:abs-trans we get that $\vdash \tau'\tau = \mathcal{L}_1 \Rightarrow \tau_1[\tau/\alpha] \restriction \mathcal{L}_2 : \star$. Finally by wf-core:weak we get $; \vdash v : \mathcal{L}_1 \Rightarrow \tau_1[\tau/\alpha] \restriction \mathcal{L}_2 \mid \Sigma$. Now by the canonical forms lemma with inversion we have three cases.
  - $v = \varnothing$. We can apply ev-core:hcolor-empty to take a step.
  - $v = \{l_1 \Rightarrow e\}$. We can apply ev-core:hcolor-sing to take a step.
  - $v = v_1 \bowtie v_2$. We can apply ev-core:hcolor-join to take a step.
- $\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\forall\beta{:}\kappa' \restriction \mathcal{L}'.\tau_1$. By Lemma C.17 and Corollary C.19 we get that $\vdash \tau'\tau = (\lambda\alpha{:}\kappa.\forall\beta{:}\kappa' \restriction \mathcal{L}'.\tau_1)\tau : \star$. Then by teq-core:abs-beta and teq-core:trans we get that $\vdash \tau'\tau = \forall\beta{:}\kappa' \restriction \mathcal{L}'.\tau_1[\tau/\alpha] : \star$. Now by wf-core:weak we have that $; \vdash v : \forall\beta{:}\kappa' \restriction \mathcal{L}'.\tau_1[\tau/\alpha] \mid \Sigma$ and by the canonical forms lemma (using the inversion we get that $v = \Lambda\beta{:}\kappa' \restriction \mathcal{L}'.e'$ and we can apply ev-core:hcolor-tabs to take a step.
- $\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\forall\iota{:}\mathrm{L}(\kappa').\tau_1$. Similar to the previous case.
- $\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\forall s{:}\mathrm{Ls}.\tau_1$. Similar to the previous case.

- Case wf-core:hout. Similar to the wf-core:hin case.
- Case wf-core:var. Can't happen.
- Case wf-core:app. Either we can take a step using ev-core:app-con1 or ev-core:app-con2, or in case the two expressions are values, we use the canonical forms lemma and see that we can apply ev-core:abs-beta.
- Case wf-core:fix. Simple application of ev-core:fix-beta.
- Case wf-core:new. Simple application of ev-core:new.
- Case wf-core:tapp. In this case either we can take a step using ev-core:tapp-con or if the expression is a value we use the canonical forms lemma and we see that we can apply the rule ev-core:tabs-beta.
- Case wf-core:lapp. Similar to the wf-core:tapp case.
- Case wf-core:sapp. Similar to the wf-core:tapp case.
- Case wf-core:out. We have $; \vdash \{\!\{e\}\!\}^{-}_{l=\tau} : \rho[\tau] \mid \Sigma$, given that $; \vdash e : \rho[l] \mid \Sigma$ and $l{:}\kappa = \tau \in \Sigma$. By induction hypothesis, if e can take a step we can apply ev-core:color-con. If $e = v$, then by the canonical forms lemma and the fact that $l \neq \ell_{\mathsf{int}}, \ell_{\rightarrow}$ because $\mathrm{dom}(()\Sigma)$ does not contain ℓ_{int} and $\ell_{\rightarrow}$ we have that $v = \{\!\{v'\}\!\}^{+}_{l=\tau}$. Then we can apply ev-core:in-out and we are done.
- Case wf-core:in. Here $; \vdash \{\!\{e\}\!\}^{+}_{l=\tau} : \rho[l] \mid \Sigma$, given that $; \vdash e : \rho[\tau] \mid \Sigma$ and $l{:}\kappa = \tau \in \Sigma$. By induction hypothesis, if e can take a step we can apply ev-core:color-con and we are done. If $e = v$ then the whole term is a value and we are done again.
- Case wf-core:weak. Follows directly from the induction hypothesis.
- Case wf-core:lindex. Straightforward application of ev-core:lindex.
- Cases wf-core:tabs, wf-core:labs, wf-core:env-empty, wf-core:env-branch, wf-core:int, wf-core:abs. They are all values already.
- Cases wf-core:env-join. In this case $; \vdash e_1 \bowtie e_2 : \mathcal{L}_1 \cup \mathcal{L}_2 \Rightarrow \tau' \restriction \mathcal{L} \mid \Sigma$, given that $; \vdash e_1 : \mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L} \mid \Sigma$ and $; \vdash e_2 : \mathcal{L}_2 \Rightarrow \tau' \restriction \mathcal{L} \mid \Sigma$. By induction hypothesis if either of e_1 and e_2 can take a step we can apply ev-core:join-con1 or ev-core:join-con2, else the whole term is a value and we are done.

□

C.7 Subject Reduction

Theorem C.51. *If* $; \vdash e : \tau \,|\, \Sigma$ *and* $\ell_{\mathsf{int}}, \ell_{\rightarrow} \notin dom(\Sigma)$ *and* $\mathcal{L}; e \mapsto \mathcal{L}'; e'$ *for* $\mathcal{L} = dom(\Sigma) \cup \{\ell_{\mathsf{int}}\} \cup \{\ell_{\rightarrow}\}$, *then* $\exists \Sigma'$ *with* $dom(\Sigma') \cup \{\ell_{\mathsf{int}}\} \cup \{\ell_{\rightarrow}\} = \mathcal{L}'$ *and* $\ell_{\mathsf{int}}, \ell_{\rightarrow} \notin dom(\Sigma')$ *such that* $; \vdash e' : \tau \,|\, \Sigma'$ *and* $\Sigma \subseteq \Sigma'$.

Proof. By induction on typing derivations for $; \vdash e : \tau \,|\, \Sigma$. The cases for wf-core:var, wf-core:int, wf-core:abs, wf-core:labs. wf-core:sabs, wf-core:empty, wf-core:env-branch cannot happen. The cases for wf-core:app, wf-core:fix, wf-core:new, wf-core:weak, wf-core:lindes, wf-core:tapp, wf-core:lapp, wf-core:sapp, wf-core:env-join follow easily using the auxilliary lemmas, the various substitution lemmas and the induction hypothesis. The most interesting cases are the following.

- Case wf-core:typecase. We have $; \vdash \mathbf{typecase}\ \tau\ e : \tau'\tau \,|\, \Sigma$, given that $; \vdash e : \mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L}_2 |\, \Sigma$, $\vdash \tau : \star$, $\vdash \tau \,|\, \mathcal{L}$, $\vdash \mathcal{L}_1 \sqsubseteq \mathcal{L}_2$, $\vdash \mathcal{L} \sqsubseteq \mathcal{L}_1$. We have two cases for evaluation of this expression.
  - Case ev-core:typecase-con. Then we have that $\mathcal{L}; e \mapsto \mathcal{L}'; e'$ and by induction hypothesis there exists an appropriate Σ' such that $; \vdash e' : \mathcal{L}_1 \Rightarrow \tau' \restriction \mathcal{L}_2 |\, \Sigma$. Then we can apply wf-core:typecase and we are done.
  - Case ev-core:typecase. Then $e = v = v_1 \bowtie \ldots \bowtie \{\ell_i^{\kappa} \Rightarrow e_1\} \bowtie \ldots v_n$, and $\vdash \tau \downarrow \rho[\ell_i^{\kappa}]$, $\rho \rightsquigarrow p$ and then $\mathcal{L}; \mathbf{typecase}\ \tau\ e \mapsto \mathcal{L}; p[e_1]$. It is enough to show that $; \vdash p[e_1] : \tau'\tau \,|\, \Sigma$. By inversion we know that $\vdash \{\ell_i^{\kappa}\} \sqsubseteq \mathcal{L}_1$, and by transitivity we get $\vdash \{\ell_i^{\kappa}\} \sqsubseteq \mathcal{L}_2$. Also by inversion it must be that $; \vdash e_1 : \tau'\langle \ell_i^{\kappa} : \kappa \restriction \mathcal{L}_2 \rangle \,|\, \Sigma$. Also we know that $\vdash \tau \,|\, \mathcal{L}$ and $\vdash \mathcal{L} \sqsubseteq \mathcal{L}_1$, so by transitivity we get $\vdash \mathcal{L} \sqsubseteq \mathcal{L}_2$. Finally
    $\vdash \tau \downarrow \rho[\ell_i^{\kappa}] = \ell_i^{\kappa}\tau_1 \ldots \tau_n$ where by inversion it must be that $\vdash \tau_i \,|\, \mathcal{L}_{\tau_i}$ with $\vdash \mathcal{L}_{\tau_i} \sqsubseteq \mathcal{L}_2$. Then we can apply Theorem C.23 to get that $; \vdash p[e_1] : \tau'\tau \,|\, \Sigma$ and we are done.
- Case wf-core:in. We have $; \vdash \{\!\{e\}\!\}^{+}_{l=\tau} : \rho[l] \,|\, \Sigma$, given that $; \vdash e : \rho[\tau] \,|\, \Sigma$ and $l{:}\kappa = \tau \in \Sigma$. We only have one case for evaluation, that is, using rule ev-core:color-con. Then $\mathcal{L}; e \mapsto \mathcal{L}'; e'$, and by induction hypothesis there exists a Σ' with $dom(\Sigma') \cup \ell_{\mathsf{int}} \cup \ell_{\rightarrow} = \mathcal{L}'$ and $\Sigma \subseteq \Sigma'$, such that $; \vdash e' : \rho[\tau] \,|\, \Sigma'$. Because $S \subseteq \Sigma'$, it must be that $l{:}\kappa = \tau \in \Sigma'$, hence by wf-core:in $; \vdash \{\!\{e'\}\!\}^{+}_{l=\tau} : \rho[l] \,|\, \Sigma'$ and we are done.
- Case wf-core:out. Here we have $; \vdash \{\!\{e\}\!\}^{-}_{l=\tau} : \rho[\tau] \,|\, \Sigma$, given that $; \vdash e : \rho[l] \,|\, \Sigma$ and $l{:}\kappa = \tau \in \Sigma$. We have the following cases for evaluation.
  - Case ev-core:color-con. In this case it must be that $\mathcal{L}; e \mapsto \mathcal{L}'; e'$, and by induction hypothesis there exists a Σ' with $dom(\Sigma') \cup \ell_{\mathsf{int}} \cup \ell_{\rightarrow} = \mathcal{L}'$ and $\Sigma \subseteq \Sigma'$, such that $; \vdash e' : \rho[l] \,|\, \Sigma'$. Because $S \subseteq \Sigma'$, it must be that $l{:}\kappa = \tau \in \Sigma'$, hence by wf-core:out $; \vdash \{\!\{e'\}\!\}^{-}_{l=\tau} : \rho[\tau] \,|\, \Sigma'$ and we are done.
  - Case ev-core:in-out. Here it must be that $\mathcal{L}; \{\!\{\{\!\{v\}\!\}^{+}_{l=\tau}\}\!\}^{-}_{l=\tau} \mapsto \mathcal{L}; v$. And we have that $; \vdash \{\!\{v\}\!\}^{+}_{l=\tau} : \rho[l] \,|\, \Sigma$ by our assumptions. By inversion then we get that $; \vdash v : \rho[\tau] \,|\, \Sigma$ and we are done.
- Case wf-core:hin. We have that $; \vdash \{\!\{e : \tau'\}\!\}^{+}_{l=\tau} : \tau'\ l \,|\, \Sigma$, given that $; \vdash e : \tau'\ l \,|\, \Sigma$ and $l{:}\kappa = \tau \in \Sigma$. We have the following cases for evaluation.
  - Case ev-core:hcolor-con. We have that $\mathcal{L}; \{\!\{e : \tau'\}\!\}^{\pm}_{l=\tau} \mapsto \mathcal{L}'; \{\!\{e' : \tau'\}\!\}^{\pm}_{l=\tau}$, given that $\mathcal{L}; e \mapsto \mathcal{L}'; e'$. Then by induction hypothesis there exists an appropriate Σ' such that $; \vdash e' : \tau'\tau \,|\, \Sigma'$ and $\Sigma \subseteq \Sigma'$ which means that $l{:}\kappa = \tau \in \Sigma'$ as well, so we can apply wf-core:hin and get the result.
  - Case ev-core:hcolor-int. Then it must the case that $\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\ell_{\mathsf{int}}$ and $e = i$. So we have that $\mathcal{L}; \mapsto \mathcal{L}; \{\!\{i : \tau'\}\!\}^{+}_{l=\tau} i$. By wf-core:int we know that $; \vdash i : \ell_{\mathsf{int}} \,|\, \Sigma$ and we can easily verify that $\vdash \tau' l = \ell_{\mathsf{int}} : \star$, which gives $; \vdash i : \tau' l \,|\, \Sigma$ by wf-core:weak and we are done if we pick the same Σ.
  - Case ev-core:color-abs1.
    We have that $\mathcal{L}; \{\!\{\lambda x{:}\sigma_1.e : \tau'\}\!\}^{+}_{l=\tau} \mapsto \mathcal{L}; \lambda x{:}(\tau_1[l/\alpha]).\{\!\{e[\{\!\{x : \lambda\alpha{:}\kappa.\tau_1\}\!\}^{-}_{l=\tau}/x] : \lambda\alpha{:}\kappa.\tau_2\}\!\}^{+}_{l=\tau}$, given

that $\vdash \tau' \downarrow \lambda\alpha{:}\kappa.\tau_1 \to \tau_2$ and $\vdash \sigma_1 = \tau_1[\tau/\alpha] : \star$. We have the following:

$$\begin{array}{rl}
 & ; \vdash \{\!\{\lambda x{:}\sigma_1.e : \tau'\}\!\}^{+}_{l=\tau} : \tau' l \mid \Sigma \quad \Rightarrow \text{ (inversion)} \\
\Rightarrow & ; \vdash \lambda x{:}\sigma_1.e : \tau'\tau \mid \Sigma \\
\Rightarrow & ; \vdash \lambda x{:}\sigma_1.e : (\lambda\alpha{:}\kappa.\tau_1 \to \tau_2)\tau \mid \Sigma \\
\Rightarrow & ; \vdash \lambda x{:}\sigma_1.e : \tau_1[\tau/\alpha] \to \tau_2[\tau/\alpha] \mid \Sigma \\
\Rightarrow & ; \vdash \lambda x{:}\sigma_1.e : \sigma_1 \to \tau_2[\tau/\alpha] \mid \Sigma \\
\Rightarrow & ; x{:}\sigma_1 \vdash e : \tau_2[\tau/\alpha] \mid \Sigma \\
\Rightarrow & ; x{:}\sigma_1 \vdash e : (\lambda\alpha{:}\kappa.\tau_2)\tau \mid \Sigma
\end{array}$$

On the other hand we have that:

$$\begin{array}{rl}
 & ; y{:}\tau_1[l/\alpha] \vdash y : \tau_1[l/\alpha] \mid \Sigma \\
\Rightarrow & ; y{:}\tau_1[l/\alpha] \vdash y : (\lambda\alpha{:}\kappa.\tau_1) l \mid \Sigma \\
\Rightarrow & ; y{:}\tau_1[l/\alpha] \vdash \{\!\{y : \lambda\alpha{:}\kappa.\tau_1\}\!\}^{-}_{l=\tau} : (\lambda\alpha{:}\kappa.\tau_1)\tau \mid \Sigma \\
\Rightarrow & ; y{:}\tau_1[l/\alpha] \vdash \{\!\{y : \lambda\alpha{:}\kappa.\tau_1\}\!\}^{-}_{l=\tau} : \sigma_1 \mid \Sigma
\end{array}$$

Now, using the previous derivation and this one, by weakening, we get:

$$\begin{array}{rll}
 & ; y{:}\tau_1[l/\alpha], x{:}\sigma_1 \vdash e : (\lambda\alpha{:}\kappa.\tau_2)\tau \mid \Sigma & \text{and by substitution lemma} \\
\Rightarrow & ; y{:}\tau_1[l/\alpha] \vdash e[\{\!\{y : \lambda\alpha{:}\kappa.\tau_1\}\!\}^{-}_{l=\tau}/x] : (\lambda\alpha{:}\kappa.\tau_2)\tau \mid \Sigma & \text{and by a renaming} \\
\Rightarrow & ; x{:}\tau_1[l/\alpha] \vdash e[\{\!\{x : \lambda\alpha{:}\kappa.\tau_1\}\!\}^{-}_{l=\tau}/x] : (\lambda\alpha{:}\kappa.\tau_2)\tau \mid \Sigma & \\
\Rightarrow & ; x{:}\tau_1[l/\alpha] \vdash \{\!\{e[\{\!\{x : \lambda\alpha{:}\kappa.\tau_1\}\!\}^{-}_{l=\tau}/x] : \lambda\alpha{:}\kappa.\tau_2\}\!\}^{+}_{l=\tau} : (\lambda\alpha{:}\kappa.\tau_2) l \mid \Sigma & \\
\Rightarrow & ; x{:}\tau_1[l/\alpha] \vdash \{\!\{e[\{\!\{x : \lambda\alpha{:}\kappa.\tau_1\}\!\}^{-}_{l=\tau}/x] : \lambda\alpha{:}\kappa.\tau_2\}\!\}^{+}_{l=\tau} : \tau_2[l/\alpha] \mid \Sigma & \\
\Rightarrow & ; \vdash \lambda x{:}\tau_1[l/\alpha].\{\!\{e[\{\!\{x : \lambda\alpha{:}\kappa.\tau_1\}\!\}^{-}_{l=\tau}/x] : \lambda\alpha{:}\kappa.\tau_2\}\!\}^{+}_{l=\tau} : \tau_1[l/\alpha] \to \tau_2[l/\alpha] \mid \Sigma & \\
\Rightarrow & ; \vdash \lambda x{:}\tau_1[l/\alpha].\{\!\{e[\{\!\{x : \lambda\alpha{:}\kappa.\tau_1\}\!\}^{-}_{l=\tau}/x] : \lambda\alpha{:}\kappa.\tau_2\}\!\}^{+}_{l=\tau} : (\lambda\alpha{:}\kappa.\tau_1 \to \tau_2) l \mid \Sigma & \\
\Rightarrow & ; \vdash \lambda x{:}\tau_1[l/\alpha].\{\!\{e[\{\!\{x : \lambda\alpha{:}\kappa.\tau_1\}\!\}^{-}_{l=\tau}/x] : \lambda\alpha{:}\kappa.\tau_2\}\!\}^{+}_{l=\tau} : \tau' l \mid \Sigma &
\end{array}$$

- Case ev-core:hcolor-base. The style of the argument is similar to one for the ev-core:hcolor-abs1 case.
- Case ev-core:hcolor-empty. The style of the argument is similar to one for the ev-core:hcolor-abs1 case.
- Case ev-core:hcolor-sing. The style of the argument is similar to one for the ev-core:hcolor-abs1 case.
- Case ev-core:hcolor-join. The style of the argument is similar to one for the ev-core:hcolor-abs1 case.
- Case ev-core:hcolor-tabs.The style of the argument is similar to one for the ev-core:hcolor-abs1 case.
- Case ev-core:hcolor-labs. The style of the argument is similar to one for the ev-core:hcolor-abs1 case.
- Case ev-core:hcolor-sabs. The style of the argument is similar to one for the ev-core:hcolor-abs1 case.

- Case wf-core:hout. The analysis for this case is similar to the wf-core:hin case.

□