ABSTRACT

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

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

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

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

1. INTRODUCTION

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

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

Currently, there are two forms of ad-hoc polymorphism in typed, functional languages. The first is based on the nominal analysis of type information, such as Haskell type classes [20]. The execution of an ad-hoc operation is determined solely by the name of the type argument (or the name of its head constructor, such as list.)

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

\[
\text{class Eq a where}
\]
\[
\text{eq} :: a -> a -> \text{Bool}
\]
\[
\text{instance Eq Int where}
\]
\[
\text{eq} x y = \text{eqint} x y
\]
\[
\text{instance Eq Bool where}
\]
\[
\text{eq} x y = \text{if } x \text{ then } y \text{ else not } y
\]
\[
\text{instance (Eq a, Eq b) \to Eq (a,b) where}
\]
\[
\text{eq} (x1,y1) (x2,y2) = \text{eq} x1 y1 && \text{eq} x2 y2
\]
\[
\text{instance (Eq a) \to Eq [a] where}
\]
\[
\]

Nominal analysis naturally limits the domain of an ad-hoc operation to those types where a definition has been provided. For example, eq is not defined for function types. Type-directed operations in a nominal framework are naturally “open”; at any time they may be extended with in-
The second form of ad-hoc polymorphism is based on the structural analysis of types. For example, intensional type analysis \[\lambda\] allows programmers to define type-directed operations by case analysis of the type structure. Polymorphic equality defined by run-time type analysis may look like the following:

\[
eq(x::a)\ (y::a) = \text{typecase a of} \begin{align*}
    \text{Int} & \rightarrow \text{eqnt x y} \\
    \text{Bool} & \rightarrow \text{if x then y else not y} \\
    (b,c) & \rightarrow \text{eq (fst x)(fst y) \&\& (snd x)(snd y)} \\
    [b] & \rightarrow \text{all2 eq x y} \\
    (b\rightarrow c) & \rightarrow \text{error "eq not defined for functions"}
\end{align*}
\]

Because type-directed operations are defined by case analysis, they are naturally “closed” to extension. In fact, cases for all types must be provided when such operations are defined. These two forms of ad-hoc polymorphism differ in the way that they treat user-defined types. User-defined types such as Haskell’s \texttt{newtype} \[\nu\], are an important part of many languages. Although these new types are isomorphic to existing types, they express application-specific distinctions that can be made by the type checker. For example, a programmer may wish to ensure that he does not confuse phone numbers with ages in an application, even though both may be represented using integers. Because nominal operations are open, they must be extended with instances for each new user-defined type. It may be tedious and even difficult to add new operations that apply to many types. Furthermore, if some types are defined in separate inaccessible modules then it is impossible for the programmer to extend the operation to those types. Instead, he must rely on the definer of the type to add the instance, and there is no guarantee that she will respect the invariants of the type-directed operation.

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

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

1.1 Combining both forms in one language

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

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

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

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

1.2 Contributions of this work

The \(\lambda_C\) language is an important step towards improving the practicality of type-directed programming. In particular, this paper has the following contributions:

- We define a language that allows the definition of both “open” and “closed” type-directed operations. Previous work has chosen one or the other, augmented with ad-hoc mechanisms to counter their difficulties.
- We define a language that allows programmers to statically restrict the domain of type-directed operations defined in a structural system in a natural manner. Previous work \[\lambda\] requires that programmers use type-level analysis or programming to makes such restrictions.
- We show how to reconcile \texttt{typecase} with the analysis of higher-order type constructors. Previous work \[\nu\] has based such analysis on the interpretation of type constructors. In \(\lambda_C\), 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 \[\lambda\] to higher-order coercions in the presence of type constructor isomorphisms.

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

2. PROGRAMMING IN \(\lambda_C\)

At the core, \(\lambda_C\) is a polymorphic lambda calculus \((F_\omega)\) \[\lambda\] augmented with type analysis and user-defined types. The syntax of \(\lambda_C\) appears in Figure 1. In addition to the standard kinds, type constructors and terms of \(F_\omega\), \(\lambda_C\) includes labels \((\ell)\) and sets of labels \((\mathcal{L})\). Labels may be considered to be “type constants” and model both built-in types (such as \texttt{int}) and user-defined types.
Kinds

\[ \kappa ::= * \mid \kappa_1 \rightarrow \kappa_2 \]

Types

\[ \tau ::= \alpha \mid \lambda : \kappa. \tau \mid \tau_1 \tau_2 \]

- \( \lambda \)-calculus
- Labels
- Type of type-poly. terms
- Type of label-poly. terms
- Type of set-poly. terms
- Type of typecase branches

Terms

\[ e ::= x \mid \lambda x : \tau. e \mid e_1 e_2 \]

- \( \lambda \)-calculus
- Integers and recursion
- Label creation
- First-order coercion
- Higher-order coercion
- Type analysis
- Branches
- Label polymorphism
- Label set polymorphism

Labels

\[ l ::= \ell_1 \mid \ell_2 \]

- Variables
- Constants

Label sets

\[ \mathcal{L} ::= s \]

- Variables
- Empty and universe
- Singleton and union

Figure 1: The core \( \lambda C \) language

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

In the following subsections, we describe the important features of \( \lambda C \) in more detail.

2.1 Generative types

The \( \lambda C \) language includes a simple mechanism for users to define new type constants. We call all type constants \( \ell \), annotated with their kinds. Some distinguished constants in this language are constructors for primitive types. The label \( \ell_0 \) is a nullary constructor for the type of integers, and \( \ell_1 \cdots \ell_n \) is the the binary constructor for function types. We use the syntactic sugar \( \ell_{\text{int}} \) and \( \ell_\rightarrow \) to refer to these two labels. However, when these labels appear in types, we use the notation \( \mid \) to stand for \( \ell_{\text{int}} \) and \( \tau_1 \rightarrow \tau_2 \) to stand for the function type \( \ell_\rightarrow \tau_1 \tau_2 \). In the examples, we extend this language with new forms of types, such as booleans (\( \text{bool} \)), products (\( \tau_1 \times \tau_2 \)), and lists (\( \ell_\text{list} \)), and add new labels constants, written \( \ell_{\text{bool}}, \ell_\times \) and \( \ell_{\text{int}} \), to form those types.

The expression \( \text{new } \ell \cdot k \) in \( e \) creates user-defined labels. This expression dynamically generates a new label constant and binds it to the label variable \( \ell \). Inside the scope \( e \), the type \( \ell \) is isomorphic to the type \( \tau \) of kind \( \kappa \).

The operators \( \{ \} \) and \( \{ \} \) coerce expressions to and from the types \( \ell \) and \( \tau \). When \( \tau \) is apparent from context we elide that annotation, as in the example below.

\[ \text{new } \ell ::= \text{int } \in \lambda x : \{ e \} \cdot \{ e \} \cdot + 3 ] \]

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

Note that even though run-time type analysis destroys the parametricity created by a type polymorphism (\( \Lambda \alpha \)), users may still hide the implementation details of abstract datatypes with these generative types. Once outside the scope of a new label, it is impossible to determine its underlying definition. For example, we know that the polymorphic function \( f \) below must treat its term argument parametrically because, even in the presence of run-time type analysis, it cannot coerce it to the type \( \text{int} \).

\[ \text{let } f = \ldots \text{ in } \]

\[ \text{new } \ell ::= \text{int } \in f [\ell_{\text{int}} \Rightarrow 1, \ell_{\text{bool}} \Rightarrow 2] \]

2.2 Type analysis with a restricted domain

The term typecase \( \tau e \) may be used to define type-directed operations in \( \lambda C \). This operator determines the head (or outermost) label of the normal form of its type argument \( \tau \), such as \( \ell_{\text{int}}, \ell_\times \) or \( \ell_{\text{int}} \). It then selects the appropriate branch from the finite map \( e \) from labels to expressions. For example, the expression typecase \( \text{int } \{ \ell_{\text{int}} \Rightarrow 1, \ell_{\text{bool}} \Rightarrow 2 \} \) evaluates to 1.

The finite map in typecase \( \tau e \) may be formed from a singleton map (such as \( \{ \ell_{\text{int}} \Rightarrow \text{int} \} \)) or the join of two finite maps \( e_1 \cdot e_2 \). In a join, if the domains are not disjoint, the second map has precedence. Compound maps such as \( \{ e_1 \Rightarrow e_1, e_2 \Rightarrow e_2 \} \cdot \{ e_3 \Rightarrow e_3 \} \) are abbreviated as \( \{ e_1 \Rightarrow e_1, e_2 \Rightarrow e_2, e_3 \Rightarrow e_3 \} \).

A challenging part of the design of \( \lambda C \) is ensuring that there is a matching branch for the analyzed type. For example, stuck expressions such as typecase \( \text{bool } \{ \ell_{\text{int}} \Rightarrow 2 \} \) should not type check, because there is no branch for the boolean type (or rather its label).

For this reason, when checking a \( \text{typecase} \) expression, \( \lambda C \) calculates the set of labels that may appear within the analyzed type and requires that set to be a subset of the set of labels that have branches in typecase. Label sets in \( \lambda C \) may be empty, \( \emptyset \), may contain a single label, \( \{ \ell \} \), or may be the union of two label sets, \( \ell_1 \cup \ell_2 \), or may be the entire universe of labels, \( \ell \). Analogously to finite maps, \( \{ \ell_1, \ldots, \ell_n \} \) abbreviates \( \{ \ell_1 \} \cup \ldots \cup \{ \ell_n \} \).

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 \( \alpha \) will be instantiated only by a type formed from the labels \( \ell_{\text{int}} \)
and \( \ell_{\text{bool}} \) (i.e., by int or bool), so \( \alpha \) will have a match in the typecase expression.

\[
\lambda x: (\{ \ell_{\text{int}}, \ell_{\text{bool}} \}). \text{typecase } \alpha \ {\ell_{\text{int}} \Rightarrow 2, \ell_{\text{bool}} \Rightarrow 3}
\]

If we annotate a type variable with \( \ell \) then it is unanalyzable because no typecase can cover all branches.\(^1\) A more realistic use of typecase is for polymorphic equality. The function eq below implements a polymorphic equality function for data objects composed of integers, booleans, products and lists. In the following examples, let \( \mathcal{L}_0 = \{ \ell_{\text{int}}, \ell_{\text{bool}}, \ell_x, \ell_{\text{int}} \} \).

\[
\text{fix eq}(\forall \alpha \times (\ell_{\mathcal{L}_0} \rightarrow \alpha \rightarrow \alpha \rightarrow \text{bool})). \quad \lambda x: (\mathcal{L}_{\mathcal{L}_0}). \text{typecase } \alpha \\
\{ \ell_{\text{int}} \Rightarrow \text{eqint,} \\
\ell_{\text{bool}} \Rightarrow \lambda x: \text{bool}. \lambda y: \text{bool}. \\
\quad \text{if } x \text{ then } y \text{ else } (not y), \\
\ell_x \Rightarrow \lambda \alpha: (\ell_{\mathcal{L}_0}, \lambda \alpha_2: (\ell_{\mathcal{L}_0}). \text{eq}(\alpha_1 \times (\lambda x: \text{int}. (\text{fst } x))(\text{fst } y)) \\
\quad & \& \text{eq}(\alpha_2, \lambda x: \text{bool}. (\text{snd } x))(\text{snd } y), \\
\ell_{\text{int}} \Rightarrow \lambda \beta: (\mathcal{L}_{\mathcal{L}_0}, \lambda x: (\text{list } \beta). \lambda y: (\text{list } \beta). \\
\quad \text{all} (\text{eq}(\beta))(x y)) \}
\]

Product types have two subcomponents so the branch for \( \ell_x \) abstracts two type variables for those subcomponents. Likewise, the \( \ell_{\text{int}} \) 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 \( \ell_{\text{int}} \) branch being used to int.

\[
\text{eq}(\text{list int}) \mapsto (\lambda \beta: (\mathcal{L}_{\mathcal{L}_0}, \lambda x: (\text{list } \beta). \lambda y: (\text{list } \beta). \\
\quad \text{all} (\text{eq}(\beta))(x y)) \ x \ y) \mapsto \lambda x: (\text{list int}). \lambda y: (\text{list int}). \text{all} 2 (\text{eq}(\text{int})) x y \]

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

### 2.3 Generative types and type analysis

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

In \( \lambda \mathcal{L} \), we provide two solutions to this problem. We can rewrite eq to be extensible with new branches for the new labels. Otherwise, we can leave eq as is and at application, coerce all the arguments to eq so that their types do not contain new labels.

#### 2.3.1 Extensible type analysis

In \( \lambda \mathcal{L} \), we can rewrite eq to be extensible with new branches for new labels. Programmers may provide new typecase branches as an additional argument to eq. The type of this argument, a first-class map from labels to expressions, is written as \( \mathcal{L}_1 \Rightarrow \tau / \mathcal{L}_2 \). The first component

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

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

\[
\Lambda s: L_{\text{a}}, \lambda y_{\text{a}}: \{s \Rightarrow (\lambda x: \alpha \rightarrow \text{string}) \{s \cup \{x\}\}\}. \\
\lambda y_{\text{a}}: \{s \Rightarrow (\lambda x: \alpha \rightarrow \text{string}) \{s \cup \{x\}\}\}. \\
\text{fix tostring, } \lambda x: \{s \cup \{x\}\}. \\
\text{typecase } \alpha \\
(\text{y}_{\alpha} : \{\text{s} \Rightarrow \{\text{s} \cup \{x\}\}\} \Rightarrow \text{string}) (\text{y}_{\alpha} : \{\text{s} \cup \{x\}\}) \Rightarrow \\
\lambda x: (\alpha \times \alpha_2). \\
\text{let } s1 = \text{if important}[s] \text{ yomp } [\alpha_1][(\text{fst } x)] \\
\text{then tostring}[\text{s}][\alpha_1][(\text{fst } x)] \\
\text{else } "\text{...}" \text{ in } \\
\text{let } s2 = \text{if important}[s] \text{ yomp } [\alpha_2][(\text{snd } x)] \\
\text{then tostring}[\text{s}][\alpha_2][(\text{snd } x)] \\
\text{else } "\text{...}" \text{ in } \\
("++ s1 ++ "," s2 ++ ")")
\]

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

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

In contrast, a `closed` polytypic operation may easily call other `closed` polytypic functions.

### 2.3.2 Higher-order coercions

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

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

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

\[
f [\text{list } \text{int}] (\text{map } (\lambda y: \{\text{y} \rightarrow \text{int}\}) x)
\]

However, operationally, this map destroys and rebuilds the

### Figure 2: Syntax necessary for the static and dynamic semantics

list, which could be computationally expensive. Higher-order coercions can coerce `x` to be of type `list int` without computational cost.

\[
f [\text{list } \text{int}] (\text{list } \text{int})^{-1}
\]

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

### 3. THE CORE LANGUAGE

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

The judgment \(\Delta, \Gamma \vdash e : \tau \Sigma\) states that a term `e` is well-formed with type `\(\tau\)` in type context `\(\Delta, \Gamma\)`, term context `\(\Sigma\)`, and possibly using type isomorphisms described by `\(\Sigma\)`. Type isomorphisms are induced by `new` expressions that introduce new label variables isomorphic to types. To show that terms are well typed often requires determining the kinds of types, with the judgment `\(\Delta \vdash \tau : \kappa\)`, and the set of possible labels that may appear in types, with the judgment `\(\Delta \vdash \tau \mid \kappa\)`. The judgment `\(\Delta, \Gamma \vdash e : \tau \mid \kappa\)` describes the small-step call-by-value operational semantics of the language. It says that a term `e` with a set of labels `\(\Delta\)` steps to a new term `e'` with a possibly larger set of labels `\(\Delta'\)`. During the evaluation of the `new` operator, the label-set component allows the selection of a fresh label that has not previously been used. In this way, it resembles an allocation semantics \([23, 24]\). The initial state of execution includes all type constants, such as `\(\ell_{\text{int}}\)` and `\(\ell_{\text{int}}\)`, in `\(\Delta\)`. The semantics for the \(\lambda\)-calculus fragment of \(\lambda_{\text{C}}\), 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:

\[
\ell_{\text{int}} \not\in \Delta \Rightarrow \\
\ell_{\text{int}} \vdash \ell_{\text{int}} = \tau \text{ in } e \rightarrow \ell_{\text{int}} \cup \{\ell_{\text{int}}\}; e[\ell_{\text{int}}/\ell_{\text{int}}]
\]
Higher-order coercions extend the expressiveness of the primitive coercions to allow the non-head positions of a type to change. As described in the last section, they are useful for coercing the types of values stored in data structures. These coercions are annotated with a type constructor \( \tau' \) that describes the part of the data structure to be coerced.

\[
\Delta, \Gamma; e : \rho[\tau] \mid \Sigma, \tau \Downarrow L; \tau \quad \Delta, \Gamma; e : \rho[l] \mid \Sigma, \tau \Downarrow L; \tau
\]

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

\[
\Delta, \Gamma; e : \tau' \Downarrow L; \tau' \quad \Delta, \Gamma; e : \tau' \Downarrow L; \tau
\]

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

\[
\Delta \vdash \tau : \star \quad \Delta \vdash \tau' \Downarrow \tau' \quad \Delta \vdash \tau' \Downarrow \tau' \Downarrow
\]

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

If \( \tau \) is a path beginning with a type variable \( \alpha \), then that is a location where a first-order coercion should be used. However, there may be other parts of the value that should be coerced (i.e., there may be other occurrences of \( \alpha \) in the path besides the head position) so inside the first-order coercion is another higher-order coercion.

Otherwise the form of \( \tau \) must match the value in the body of the coercion. For each form of value there is an operational rule. For example, if \( \tau \) 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 \( \text{typecase} \) is below:

\[
\Delta \vdash \tau \Downarrow \rho[\alpha] \quad \{ \rho[\alpha] \Rightarrow e' \} \in v \quad \rho \Rightarrow p
\]

Thus, \( \text{typecase} \) 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 \( p \) in an obvious fashion.

The static semantics of \texttt{typcase} is defined by the following rule.

\[
\frac{\Delta \vdash \tau : \star \quad \Delta; \Gamma \vdash e : \mathcal{L}\left[L\right] \Sigma}{\Delta; \Gamma \vdash \texttt{typcase} \quad \tau \, : \, \tau' \left( \mathcal{L} \right) \quad \Sigma}
\]

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

\[
\frac{\alpha ; \kappa \mid \mathcal{L} \in \Delta}{\Delta \vdash \alpha \mid \mathcal{L}} \quad \frac{\alpha ; \kappa \in \Delta}{\Delta \vdash \alpha \mid \emptyset}
\]

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 \( \lambda \Sigma \) 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 \texttt{typcase} by not including rules to determine a label set for those types.

Once the rule for type checking \texttt{typcase} 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 \( \tau \) that mentions labels in \( \Delta \), this rule checks that the map can handle all possible labels in \( \tau \) with \( \mathcal{L} \subseteq \mathcal{L}_1 \).

The result type of \texttt{typcase} depends on the type of the map argument, \( \mathcal{L}_1 \Rightarrow \tau \mid \mathcal{L}_2 \). The most important rule for checking maps is the rule for singleton maps below.

\[
\frac{\Delta \vdash \mathcal{L} : \mathcal{L}_2 \quad \Delta \vdash l : \mathcal{L} \left( \kappa \right) \quad \Delta; \Gamma \vdash e : \tau' \left( l : \kappa \mid \mathcal{L} \right) \mid \Sigma}{\Delta; \Gamma \vdash \left\{ l \Rightarrow e \right\} : \left\{ \mathcal{L} \right\} \Rightarrow \tau' \mid \mathcal{L} \mid \Sigma}
\]

The first component of the map type (in this case \( l \)) describes the domain of the map and the second two components (\( \tau' \) and \( \mathcal{L}' \)) describe the types of the branches of the map. The judgments \( \Delta \vdash l : \mathcal{L} \left( \kappa \right) \) and \( \Delta \vdash \mathcal{L} : \mathcal{L}_s \) ensure that the label \( l \) and label set \( \mathcal{L} \) are well-formed with respect to the type context \( \Delta \). For labels of higher kind, \texttt{typcase} 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 \( \tau' \left( \mathcal{L} \mid \kappa \right) \). This notation is defined by the following rules:

\[
\tau' \left( \mathcal{L} \mid \kappa \right) \quad \equiv \quad \tau' \quad \tau' \left( \tau \mid \kappa_1 \Rightarrow \kappa_2 \mid \mathcal{L} \right) \quad \equiv \quad \forall \alpha : \kappa_1 \mid \mathcal{L} . \tau' \alpha : \kappa_2 \mid \mathcal{L}
\]

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 \texttt{typcase} 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 \texttt{typcase} rule conservatively determines the set of labels that may occur \texttt{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 \texttt{typcase}. 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 cata-morphisms 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 \( \lambda \Sigma \) language is type sound, following from the usual progress and preservation theorems \([32]\). The proofs of these theorems are inductions over the derivations defined.

Theorem 3.1 (Progress). If \( \epsilon_{\text{int}}, \epsilon_{\text{...}} \notin \text{dom}(\Sigma) \) and \( \vdash e : \tau \mid \Sigma \), then \( e \) is value, or if \( \mathcal{L} = \text{dom}(\Sigma) \cup \{ \epsilon_{\text{int}}, \epsilon_{\text{...}} \} \), then there exist some \( \mathcal{L}' \), \( e' \) such that \( \mathcal{L} ; e \Rightarrow \mathcal{L}' ; e' \).

Theorem 3.2 (Preservation). If \( \epsilon_{\text{int}}, \epsilon_{\text{...}} \notin \text{dom}(\Sigma) \) and \( \vdash e : \tau \mid \Sigma \) and \( \mathcal{L} ; e \Rightarrow \mathcal{L}' ; e' \) if \( \mathcal{L} = \text{dom}(\Sigma) \cup \{ \epsilon_{\text{int}}, \epsilon_{\text{...}} \} \), then there exists \( \Sigma' \), with \( \mathcal{L}' = \text{dom}(\Sigma') \cup \{ \epsilon_{\text{int}}, \epsilon_{\text{...}} \} \), such that \( \vdash e' : \tau \mid \Sigma' \) and \( \Sigma \subseteq \Sigma' \).

We have also shown that the coercions are not necessary to the operational semantics. An untyped calculus where the coercions have been erased (preserving types for analysis) has the same operational behavior as this calculus. In other words, expressions in \( \lambda \Sigma \) evaluate to a value if and only if their coercion-erased versions evaluate to the coercion-erased value.

4. Full Reflexivity

The core language demonstrates the basic idea of extensible \texttt{typcase} expressions, but does not offer the capability of full reflexivity. Some types cannot be analyzed by \texttt{typcase}. The full \( \lambda \Sigma \) language addresses this problem and extends the set of analyzable types to include all types. For more expressiveness, the full language also includes label and label set runtime analysis operators. In the rest of this section we discuss these extensions. The modifications to
Kinds $\kappa ::= \chi | \ast | \kappa_1 \rightarrow \kappa_2 | L(\kappa_1) \rightarrow \kappa_2 | Ls \rightarrow \kappa | \forall \chi.\kappa$

Labels $l ::= \ldots$

Label sets $\mathcal{L} ::= \ldots$

Types $\tau ::= \alpha | l | \lambda \kappa.\tau | \tau_1 \tau_2 | \lambda \xi:\lambda(\kappa).\tau | \tau_i | \tau(\tau: \kappa | \mathcal{L})$

Terms $e ::= \ldots | \text{setcase } \mathcal{L} \theta | \text{lindex } l$

Figure 4: Modifications for full reflexivity

Figure 5: Distinguished label kinds

\begin{align*}
\ell_{\text{int}} & : \ast & \text{integers} \\
\ell_{\rightarrow} & : \ast \rightarrow \ast \rightarrow \ast & \text{function type creator} \\
\ell_{\rightarrow} & : \forall \chi.(\lambda \rightarrow \ast) \rightarrow Ls \rightarrow \ast & \text{type polymorphism} \\
\ell_{\rightarrow} & : \forall \chi.(\lambda L(\chi) \rightarrow \ast) \rightarrow \ast & \text{label polymorphism} \\
\ell_{\rightarrow} & : (Ls \rightarrow \ast) \rightarrow \ast & \text{label set polymorphism} \\
\ell_{\rightarrow} & : (\forall \chi.\ast) \rightarrow \ast & \text{kind polymorphism} \\
\ell_{\map} & : Ls \rightarrow (\ast \rightarrow \ast) \rightarrow Ls \rightarrow \ast & \text{map type}
\end{align*}

Figure 6: Polykinded type equivalences

\begin{align*}
\tau(\tau: \kappa | \mathcal{L}) & \equiv \tau' \tau \\
\tau(\tau: \kappa_1 \rightarrow \kappa_2 | \mathcal{L}) & \equiv \forall \alpha : \mathcal{L}.\tau'(\tau \alpha: \kappa_2 | \mathcal{L}) \\
\tau(\tau: L(\kappa_1) \rightarrow \kappa_2 | \mathcal{L}) & \equiv \forall l_1 : L(\kappa_1).\tau'(\tau l_1: \kappa_2 | \mathcal{L}) \\
\tau(\tau: Ls \rightarrow \kappa | \mathcal{L}) & \equiv \forall \lambda \chi:\forall \chi.(\lambda \chi.\kappa | \mathcal{L}).\tau' \equiv \forall \lambda \chi:\forall \chi.(\lambda \chi.\kappa | \mathcal{L}).\tau'
\end{align*}

Figure 7: Syntactic sugar for types

\begin{align*}
\text{int} & \equiv \ell_{\text{int}} \\
\tau_1 \rightarrow \tau_2 & \equiv \ell_{\rightarrow} \tau_1 \tau_2 \\
\forall \alpha : \mathcal{L}.\tau & \equiv \ell_{\rightarrow} \forall \alpha : \mathcal{L}.\tau \\
\forall \chi.\tau & \equiv \ell_{\rightarrow} (\lambda \chi.\tau) \\
\forall \lambda \chi:\forall \chi.(\lambda \chi.\kappa | \mathcal{L}).\tau & \equiv \ell_{\rightarrow} (\lambda \chi:\forall \chi.(\lambda \chi.\kappa | \mathcal{L}).\tau)
\end{align*}

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

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

\begin{align*}
\text{fix } \text{settostring} : \forall \alpha : \mathcal{L}.\text{string.}\alpha.\mathcal{L}.
\text{setcase } \alpha \\
& \{ \emptyset \rightarrow \text{"\"} \\
& \cup \rightarrow \alpha.\mathcal{L}_1.\alpha.\mathcal{L}_2.\mathcal{L}_s. \\
& \{ \text{settostring} (\mathcal{L}_1) \} \uplus \{ \text{settostring} (\mathcal{L}_2) \} \\
& \{ \rightarrow \alpha.\forall \chi.:\forall \chi.(\lambda \chi.\mathcal{L}_s.\text{int2string}(\text{lindex}(i))) \\
& \mathcal{U} \rightarrow \text{"\{"} \\
& \} \\
\}
\end{align*}

The rule to type check \textbf{setcase} is below.

\begin{align*}
\Delta \vdash \tau : Ls \rightarrow \ast & \\
\Delta; \Gamma \vdash e_0 : \tau \emptyset \Sigma & \Delta; \Gamma \vdash e_1 : \forall \chi.\forall \chi.(\lambda \chi.\tau(\tau i : \mathcal{L}) | \mathcal{L}) \Sigma \\
\Delta; \Gamma \vdash e_0 : \forall \lambda \chi:\forall \chi.(\lambda \chi.\kappa) \rightarrow e_1 | \Sigma \\
\Delta; \Gamma \vdash e_0 : e_1 \mathcal{U} \Sigma & \Delta \vdash e_0 : \tau \mathcal{U} | \Sigma \\
\Delta \vdash \text{setcase } \mathcal{L} ; \{ \emptyset \rightarrow e_0, \{ \} \rightarrow e_1, \\
& \cup \rightarrow e_0, \mathcal{U} \rightarrow e_1 \} : \tau' \mathcal{L} | \Sigma
\end{align*}

In this rule, $e_1$ must be able to take any label as its argument, whatever the kind of the label. Therefore this expression must be kind polymorphic.

5. **HIGHER-ORDER ANALYSIS**

Higher-order type analysis \cite{higher-order} is an extension of run-time analysis to types of higher-order kind. It is used to define
operations in terms of parameterized data structures, such as lists and trees. These operations must be able to distinguish between the type parameter and the rest of the type. For example, a generic “length” operation that determines the length of a list and the number of nodes in a tree must be able to distinguish between the data (no matter what type it is) and the rest of the structure. Because $\lambda c$ can generate new labels at run time, it can make such distinctions.

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

However, in $\lambda c$, analysis is over the weak-head normal form of types. Because equal types have the same normal form, such equivalences are already preserved. Furthermore, because we know that all constructors of kind $\ast \rightarrow \ast$ are equivalent to type functions (by extensionality) we can encode the analysis of such a constructor as a polymorphic term function, whose body uses typecase to analyze a constructor of kind $\ast$.

For example, suppose $f$ is an open polytypic operation of type $\forall x : \text{LS} : \forall \beta : (\chi \cdot \ast) \cdot (s \Rightarrow \tau' \cdot s \cup \ell) \rightarrow \tau' \cdot \beta$. Say we want to use the instance of $f$ for the type $\tau$ of kind $\ast \rightarrow \ast$, and that $\ell$ contains all the labels in $\tau$. To do so, we modify the call site of $f$ to be a polytypic function, because that is the interpretation of type functions. This function abstracts the type argument $\beta$ and a branch $x$ as the interpretation of $\beta$. It then creates a new label for $\beta$ and passes a branch to $f$ that maps the new label to the interpretation of $\beta$.

That way, no matter what type $\beta$ is instantiated with, its interpretation will always be $x$.

\[
\lambda \beta \cdot \| \ell, \lambda c \cdot \tau' \cdot \beta, \text{new} \; c : \ast = \beta \; \text{in} \; \| \ell \{ \{ l \} \quad \{ r \} \mid \ell \Rightarrow \| x : \tau' \| \ell \} : \tau' \| \ell \}
\]

6. EXTENSIONS

Default branches. One difficulty of working with $\lambda c$ 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 $\{ l \Rightarrow e \}$ with a domain of all labels. With this extension, type variables restricted by $\mathcal{U}$ are not parametric.

\[
\Delta, \Gamma \vdash e : \forall x \forall \alpha : \chi \cdot \| \mathcal{U} \rightarrow (\alpha : \chi \cdot \| \mathcal{U}) \mid \Sigma
\]

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 $\lambda c$ 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 un coercions. New types in $\lambda c$ 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 enroll 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 $\lbrack \cdot \rbrack_{\ell}$ for this uncoercing.

\[
\Delta, \Gamma \vdash e : \tau' \mid \Sigma \quad \ell \cdot k = \tau \in \Sigma \quad \Delta, \Gamma \vdash \ell \cup \{ l \} \mid \Sigma
\]

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

\[
\lbrack \lbrack \text{inr} (1), \lbrack \text{inr} (3), \lbrack \text{inl} (\iota) \rbrack \rbrack \rbrack \rbrack, \lambda c \ast \cdot \alpha \mid \ell, \ell +, \ell +, \ell + \rbrack_{\iota} \ast
\]

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 $\ell$.

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. Because labels are an integral part of $\lambda c$, 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 labels to types of kind $\ast$. Finite type maps are new syntactic category with their own form of abstraction and application in both the type and term languages, as well as finite map analysis. Rules analogous to those for label set subsumption, membership and equality can be defined for these finite maps.

\[
\begin{align*}
M & ::= \emptyset \mid \{ l : \tau \mid m \} \cdot M_1 \cup M_2 \\
\kappa & ::= \ldots \mid \text{Map} \rightarrow \kappa \\
\tau & ::= \ldots \mid \lambda m . \text{Map} \cdot \tau \cdot M \\
e & ::= \ldots \mid \text{Map} \cdot \text{Map} \cdot e \mid e[M] \mid \text{mapcase} \; M \; \phi \\
\phi & ::= \{ e \Rightarrow e_1 \} \mid \{ e \Rightarrow e_2 \} \mid \bigcup \{ e \}
\end{align*}
\]

The distinguished label $\ell_{\text{rec}}$ of kind $(\text{Map} \rightarrow \ast)$ 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 $\emptyset$, 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 $\ast$. If arbitrarily-kindred labels were allowed, then code analyzing record types would need to be kind polymorphic, limiting its usefulness.
7. RELATED WORK

There is much research on type-directed programming. Run-time type analysis allows the structural analysis of dynamic type information. Abadi, et al. introduced a type "dynamic" to which types could be coerced, and later via case analysis, extracted [1]. The core semantics of typecase in $\lambda_C$ is similar to the intensional polymorphism of Harper and Morrisett [11]. However, $\lambda_C$ does not include a type-level analysis operator. Our extension of $\lambda_C$ to be fully reflexive follows a similar extension of Harper and Morrisett’s language by Trifonov, Saha, and Shao [25]. Weirich [31] extended run-time analysis to higher-order type constructors following the work of Hinze [12].

Generic programming uses the structure of datatypes to generate specialized operations at compile time. The Charity language [3] automatically generates folds for datatypes. PolyP [15] is an extension of Haskell that allows the definition of polytypic operations based on positive, regular datatypes. Functorial ML [17] bases polytypic operations on the composition of functors, and has lead to the programming language Flish [16]. Generic Haskell [8], following the work of Hinze [12] explores an extension to automatically derive type class instances by looking at the underlying structure of new types. Dependency-style Generic Haskell [20] revises the Generic Haskell language to be based on the names of types instead of their structure. However, to automatically define more generic functions, it converts user-defined types into their underlying structural representations if a specific definition has not been provided.

Many languages use a form of generative types to represent application-specific abstractions. For example, Standard ML [21] and Haskell [23] rely on datatype generativity in type inference. Modern module systems also provide generative types [5]. When the definition of the new type is known, the type isomorphisms of this paper differ from calculi with type equalities (such as provided by Harper and Lillibridge [10] or Stone and Harper [27]). In that type they require explicit terms to coerce between a type name and its definition. While explicit coercions are more difficult for the programmer to use, they simplify the semantics of the generative types. Explicit coercions also make sense in conjunction with type-directed programming because even if the definition is known, the distinction should still be made during dynamic type analysis.

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

8. DISCUSSION

In conclusion, the $\lambda_C$ language provides a good way to understand the properties of both nominal and structural type analysis. Because it can represent both forms, it makes apparent the advantages and disadvantages of each. We view $\lambda_C$ as a solid foundation for the design of a user-level language that incorporates both versions of polytypism.

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

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

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

Furthermore, it is also important to add type-level analysis of types to the language. As shown in past work, it is impossible to assign types to some type-directed functions without this feature. One way to do so might be to extend the primitive-recurcive operator of Trifonov et al. [25] to include first-class maps from labels to types.
Acknowledgments

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

9. REFERENCES


APPENDIX

A. LANGUAGE

A.1 Syntax

Kinds $\kappa ::= \chi \mid \tau \mid \kappa_1 \rightarrow \kappa_2 \mid \forall \chi.\kappa$

Labels $l ::= \ell_1 \mid \ell_2$

Label sets $\mathcal{L} ::= \emptyset \mid \{ l \} \mid \mathcal{L}_1 \cup \mathcal{L}_2 \mid \mathcal{U}$

Types $\tau ::= \alpha \mid \lambda \alpha.\tau \mid \tau_1 \tau_2 \mid \alpha \cdot \mathcal{L}(\alpha) \cdot \tau \mid \tau \mathcal{I}$

Terms $e ::= x \mid \alpha : \tau. e \mid \mathbf{fix} \, x : \tau . e \mid i$

| new $\lambda \nu = \tau \; e \mid \{ e : \tau \}_{\nu=t}^{\tau_1}$
| typecase $\tau \; e \mid \mathbf{setcase} \; \ell \; \theta$

Setcase $\theta ::= \{ \ell \Rightarrow e_{\ell} \mid \nu \Rightarrow \nu \} \Rightarrow e_{\nu} \cup e_{\ell} \cup e_{\ell}$

A.2 Judgments

Static Judgments

Kind well-formedness $\Delta \vdash \kappa$
Label well-formedness $\Delta \vdash \ell : \mathcal{L}(\kappa)$
Label set well-formedness $\Delta \vdash \mathcal{L} : \mathcal{L}_s$
Label set subsumption $\Delta \vdash \mathcal{L}_1 \subseteq \mathcal{L}_2$
Label set equivalence $\Delta \vdash \mathcal{L}_1 = \mathcal{L}_2$
Type well-formedness $\Delta \vdash \tau : \kappa$
Type label set analysis $\Delta \vdash \tau \mathcal{L}_s = \tau_2 : \kappa$
Type equivalence $\Delta \vdash \tau_1 = \tau_2 : \kappa$
Term well-formedness $\Delta; \Gamma \vdash e : \tau \mid \Sigma$

Dynamic Judgments

Small-step evaluation $\mathcal{L}; e \rightsquigarrow \mathcal{L}'; e'$
Weak-head reduction $\Delta \vdash \tau \downarrow \tau'$
Weak-head normalization $\Delta \vdash \tau \Rightarrow \tau'$
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

$$\Delta \vdash \chi \in \Delta \quad \text{wfk:var} \quad \Delta \vdash \tau \in \Delta \quad \text{wfk:var}$$

$$\Delta \vdash \kappa \quad \Delta \vdash \kappa_1 \rightarrow \kappa_2 \quad \text{wfk:arrow}$$

$$\Delta \vdash \mathcal{L}(\kappa) \rightarrow \mathcal{L}_2 \quad \text{wfk:arrow}$$

$$\Delta \vdash \forall \chi.\kappa \quad \Delta \vdash L \quad \text{wfk:arrow}$$

A.3.2 Label well-formedness

$$\Delta \vdash \ell \in \Delta \quad \text{wfk:const} \quad \Delta \vdash \ell : \mathcal{L}(\kappa) \quad \text{wfk:var}$$

$$\Delta \vdash \emptyset : \mathcal{L}_s \quad \text{wfls:empty} \quad \Delta \vdash \{ l \} : \mathcal{L}_s \quad \text{wfls:sing}$$

$$\Delta \vdash \mathcal{L}_1 \cup \mathcal{L}_2 : \mathcal{L}_s \quad \text{wfls:union}$$

$$\Delta \vdash \mathcal{U} : \mathcal{L}_s \quad \text{wfls:univ}$$

A.3.3 Label set well-formedness

$$\Delta \vdash \alpha : \kappa \quad \text{twf:var} \quad \Delta \vdash \alpha \cdot \mathcal{L}(\alpha) \cdot \kappa \quad \text{twf:var}$$

$$\Delta \vdash \tau_1 : \kappa_1 \rightarrow \kappa_2 \quad \text{twf:app} \quad \Delta \vdash \tau_1 \rightarrow \tau_2 : \kappa_1 \rightarrow \kappa_2 \quad \text{twf:polyk}$$

$$\Delta \vdash \tau_1 : \kappa_1 \rightarrow \kappa_2 \quad \Delta \vdash \tau \rightarrow \tau' : \star \rightarrow \star \quad \Delta \vdash \mathcal{L} : \mathcal{L}_s \quad \text{twf:app}$$

$$\Delta \vdash \ell : \mathcal{L}(\kappa) \quad \text{twf:var} \quad \Delta \vdash \ell : \kappa \quad \text{twf:var}$$
\[
\Delta \vdash \tau \colon \kappa \\
\Delta, \chi \vdash \tau \colon \forall \chi.\kappa \\
\text{twf:kabs}
\]
\[
\Delta \vdash \tau \colon \forall \chi.\kappa, \kappa_2 \\
\Delta \vdash \tau_1 \\
\text{twf:kapp}
\]
\[
\Delta, \iota \vdash L(\kappa_1) \vdash \tau \colon \kappa_2 \\
\Delta \vdash \tau \colon \forall \chi.\kappa_1 \vdash L(\kappa_1) \rightarrow \kappa_2 \\
\text{twf:labs}
\]
\[
\Delta \vdash \tau \colon L(\kappa_1) \rightarrow \kappa_2 \\
\Delta \vdash \tau_1 \colon \kappa_2 \\
\Delta \vdash \tau \colon L(\kappa_1) \\
\Delta \vdash \iota \colon L(\kappa_1) \\
\text{twf:flapp}
\]
\[
\Delta, \iota \vdash L(\kappa_1) \vdash \tau \colon \kappa_2 \\
\Delta \vdash \tau_1 \colon \kappa_2 \\
\Delta \vdash \tau \colon L(\kappa_1) \\
\Delta \vdash \iota \colon L(\kappa_1) \\
\text{twf:sapp}
\]
\[
\Delta, \iota \vdash L(\kappa_1) \vdash \tau \colon \kappa_2 \\
\Delta \vdash \tau \colon L(\kappa_1) \\
\Delta \vdash \iota \colon L(\kappa_1) \\
\text{twf:sapp}
\]

A.3.7 Label set equivalence

\[
\Delta \vdash L_1 \subseteq L_2 \\
\Delta \vdash L_1 \subseteq L_2 \\
\Delta \vdash L_1 = L_2 \\
\text{seq:deriv}
\]

A.3.8 Type equivalence

\[
\Delta \vdash \tau \vdash \tau \colon \kappa \\
\Delta \vdash \tau_1 = \tau_2 : \kappa \\
\Delta \vdash \tau_1 = \tau_2 : \kappa \\
\text{teq:refl}
\]
\[
\Delta \vdash \tau_1 = \tau_2 : \kappa \\
\Delta \vdash \tau_1 = \tau_2 : \kappa \\
\Delta \vdash \tau_1 = \tau_2 : \kappa \\
\text{teq:sym}
\]
\[
\Delta \vdash \tau_1 = \tau_2 : \kappa \\
\Delta \vdash \tau_1 = \tau_2 : \kappa \\
\Delta \vdash \tau_1 = \tau_2 : \kappa \\
\text{teq:trans}
\]
\[
\Delta \vdash \lambda \alpha.\kappa_1.\kappa_2 \vdash \tau_1 \rightarrow \kappa_2 \\
\Delta \vdash \lambda \alpha.\kappa_1.\kappa_2 \vdash \tau_2 \rightarrow \kappa_2 \\
\Delta \vdash \lambda \alpha.\kappa_1.\kappa_2 \vdash \tau_1 \rightarrow \kappa_2 \\
\text{teq:abs-beta}
\]
\[
\Delta \vdash \tau = \tau \vdash \tau : \kappa \\
\Delta \vdash \tau = \tau : \kappa \\
\Delta \vdash \tau = \tau : \kappa \\
\text{teq:abs-eta}
\]
\[
\Delta \vdash \tau \vdash \tau : \kappa \\
\Delta \vdash \tau \vdash \tau : \kappa \\
\Delta \vdash \tau \vdash \tau : \kappa \\
\text{teq:app-con}
\]
\[
\Delta \vdash \tau \vdash \tau : \kappa \\
\Delta \vdash \tau \vdash \tau : \kappa \\
\Delta \vdash \tau \vdash \tau : \kappa \\
\text{teq:app-con}
\]
\[
\Delta \vdash \tau \vdash \tau : \kappa \\
\Delta \vdash \tau \vdash \tau : \kappa \\
\Delta \vdash \tau \vdash \tau : \kappa \\
\text{teq:app-con}
\]

A.3.6 Label set subsumption

\[
\Delta \vdash L_1 \subseteq L_2 \\
\Delta \vdash L_1 \subseteq L_2 \\
\Delta \vdash L_1 \subseteq L_2 \\
\text{ss:refl}
\]
\[
\Delta \vdash L_1 \subseteq L_2 \\
\Delta \vdash L_1 \subseteq L_2 \\
\Delta \vdash L_1 \subseteq L_2 \\
\text{ss:trans}
\]
\[
\Delta \vdash L_1 \subseteq L_2 \\
\Delta \vdash L_1 \subseteq L_2 \\
\Delta \vdash L_1 \subseteq L_2 \\
\text{ss:union-left}
\]
\[
\Delta \vdash L_1 \subseteq L_2 \\
\Delta \vdash L_1 \subseteq L_2 \\
\Delta \vdash L_1 \subseteq L_2 \\
\text{ss:union-right1}
\]
\[
\Delta \vdash L_1 \subseteq L_2 \\
\Delta \vdash L_1 \subseteq L_2 \\
\Delta \vdash L_1 \subseteq L_2 \\
\text{ss:union-right2}
\]
\[
\Delta \vdash L : L_2 \\
\Delta \vdash \bot \not\subseteq L \\
\Delta \vdash \bot \not\subseteq L \\
\text{ss:empty}
\]
\[
\Delta \vdash L : L_2 \\
\Delta \vdash \bot \not\subseteq L \\
\Delta \vdash \bot \not\subseteq L \\
\text{ss:univ}
\]
A.3.9 Term well-formedness
A.4 Dynamic semantics

Values
\[ v := \lambda x : \sigma. e, \{ e \} \uplus i | i \]
\[ \varnothing | \{ l \Rightarrow e \} \cup v_1 \cup v_2 \]
\[ \Delta : \alpha \vdash \{ \lambda \alpha : \tau. e \} \]
\[ \Delta : \alpha \vdash \lambda \alpha : \tau. e \]

Tycon paths
\[ p ::= \bullet | p \tau | p \ell | p [\ell] | p [\ell] \]

Term paths
\[ p ::= \bullet | p \tau | p \ell | p [\ell] | p [\ell] \]

A.4.1 Weak-head reduction for types

\[ \Delta \vdash (\lambda \alpha : \kappa. \tau_2) \Downarrow \tau_1[\tau_2/\alpha] \] \text{whr:abs-beta} \\
\[ \Delta \vdash (\lambda \alpha : \kappa. \tau) \Downarrow \tau / \alpha \] \text{whr:abs-con} \\
\[ \Delta \vdash \tau_1 \Downarrow \tau' \] \text{whr:app-con} \\
\[ \Delta \vdash (\lambda : \ell. \kappa) \Downarrow \tau \Downarrow \tau' \] \text{whr:labs-beta} \\
\[ \Delta \vdash \tau \Downarrow \tau' \] \text{whr:app-con} \\

\[ \Delta \vdash (\lambda s : \ell s. \tau) \Downarrow \tau[\ell / s] \] \text{whr:sabs} \\
\[ \Delta \vdash \tau \Downarrow \tau' \] \text{whr:app-con} \\
\[ \Delta \vdash (\lambda \chi. \tau)[\kappa] \Downarrow \tau[\kappa / x] \] \text{whr:kapp-beta} \\
\[ \Delta \vdash \tau \Downarrow \tau' \] \text{whr:kapp-con}

A.4.2 Weak-head normalization for types

\[ \Delta \vdash \tau \Downarrow \tau' \] \text{whr:polyk-type} \\
\[ \Delta \vdash \tau \Downarrow \tau' \] \text{whr:polyk-a} \\
\[ \Delta \vdash \tau \Downarrow \tau' \] \text{whr:polyk-la} \\
\[ \Delta \vdash \tau \Downarrow \tau' \] \text{whr:polyk-sa} \\
\[ \Delta \vdash \tau \Downarrow \tau' \] \text{whr:polyk-all}

A.4.3 Reduction for label sets

\[ L_1 \cup L_2 = L_1 \cup L_2 \] \text{lsr:union-conj} \\
\[ L_1 \cup L_2 = L_1 \cup L_2 \] \text{lsr:union-conj} \\
\[ \emptyset \cup L \downarrow L \] \text{lsr:union-empty} \\
\[ U \cup L \downarrow U \] \text{lsr:union-univ} \\
\[ \forall \{ \ell^i \} \subseteq L : i < j \] \text{lsr:union-swap} \\
\[ (L_1 \cup L_2) \cup L_3 = L_1 \cup (L_2 \cup L_3) \] \text{lsr:union-assoc} \\
\[ L_1 \cup L_2 = L_2 \cup L_1 \] \text{lsr:union-comm} \\

A.4.4 Label set normal forms

\[ \emptyset \norm \text{In:empty} \] \\
\[ U \norm \text{In:univ} \] \\
\[ \{ \ell^i \} \norm \text{In:sing} \] \\
\[ \forall \ell^i \in L, i < j \] \text{In:union}

A.4.5 Path conversion
A.4.6 Computation rules

\[
\begin{align*}
\mathcal{L}; (\lambda x : \tau . e) v_2 &\mapsto \mathcal{L}; e[v_2/x]\quad \text{ev:abs-beta} \\
\mathcal{L}; \mathtt{fix} \ x . \tau . e &\mapsto \mathcal{L}; e[\mathtt{fix} \ x . \tau . e/x]\quad \text{ev:fix-beta} \\
\mathcal{L}; (\Lambda \alpha : \Sigma . e) \tau &\mapsto \mathcal{L}; e[\tau/\alpha]\quad \text{ev:tabs-beta} \\
\mathcal{L}; (\Lambda L . (\kappa . e)) \ell &\mapsto \mathcal{L}; e[\ell/\ell]\quad \text{ev:labs-beta} \\
\mathcal{L}; (\Lambda s . L . e) \ell &\mapsto \mathcal{L}; e[\ell/s]\quad \text{ev:sabs-beta} \\
\mathcal{L}; (\Lambda X . e) \kappa &\mapsto \mathcal{L}; e[\kappa/X]\quad \text{ev:kabs-beta} \\
\mathcal{L}; \{v\}^{\top}_{\ell \mathcal{L} \tau} &\mapsto \mathcal{L}; v\quad \text{ev:in-out} \\
\mathcal{L}; \text{new} \ e = \tau \in \ell &\mapsto \mathcal{L} \cup \{\ell_i; e[\ell_i/e]\}\quad \text{ev:new}
\end{align*}
\]

\[\vdash \tau' \downarrow \Lambda \alpha : \kappa . \rho[\alpha] \quad \text{ev:hc-base}
\]
\[\mathcal{L}; \{v: \tau\}^{\top}_{\ell \mathcal{L} \tau} \mapsto \mathcal{L}; \{v: \Lambda \alpha : \kappa . \rho[\tau]\}^{\top}_{\ell \mathcal{L} \tau} \quad \text{ev:hc-base-out}
\]
\[\vdash \tau' \downarrow \Lambda \alpha : \kappa . \ell \tau \quad \text{ev:hc-int}
\]
\[\mathcal{L}; \{v: \tau\}^{\top}_{i \mathcal{L} \tau} \mapsto \mathcal{L}; \{v: \Lambda \alpha : \kappa . \ell[\tau/\tau]\}^{\top}_{i \mathcal{L} \tau} \quad \text{ev:hc-base-out}
\]
\[\vdash \tau' \downarrow \Lambda \alpha : \kappa . \tau_1 \rightarrow \tau_2
\]
\[\vdash \tau_1 = \tau_2[\tau/\tau] : * \quad \text{ev:hc-a1}
\]
\[\mathcal{L}; \{v : \tau\}^{\top}_{\ell \mathcal{L} \tau} \mapsto \mathcal{L}; (\Lambda \alpha : \tau) . \ell[\tau/\tau]\quad \text{ev:hc-a1}
\]
\[\mathcal{L}; \{v : \tau\}^{\top}_{i \mathcal{L} \tau} \mapsto \mathcal{L}; (\Lambda \alpha : \tau) . \ell[\tau/\tau]\quad \text{ev:hc-a1}
\]

A.4.7 Congruence rules

\[\mathcal{L} \downarrow \mathcal{L} \quad \text{ev:index}
\]
\[
\begin{align*}
&\mathcal{L}; e_1 \rightarrow \mathcal{L}'; e'_1 \quad \text{ev:app-con1} \\
&\mathcal{L}; e_1 e_2 \rightarrow \mathcal{L}'; e'_1 e'_2 \quad \text{ev:app-con2} \\
&\mathcal{L}; e \rightarrow \mathcal{L}'; e' \\
&\mathcal{L}; e v \rightarrow \mathcal{L}'; e' v' \quad \text{ev:color-con} \\
&\mathcal{L}; \{ \{ e \} \} \pm l = \tau \rightarrow \mathcal{L}' \{ \{ e' \} \} \pm l = \tau \quad \text{ev:color-con} \\
&\mathcal{L}; e_1 \rightarrow \mathcal{L}'; e'_1 \\
&\mathcal{L}; e_1 \& e_2 \rightarrow \mathcal{L}'; e'_1 \& e'_2 \quad \text{ev:join-con1} \\
&\mathcal{L}; e_1 \rightarrow \mathcal{L}'; e'_1 \\
&\mathcal{L}; e_2 \rightarrow \mathcal{L}'; e'_2 \quad \text{ev:join-con2} \\
&\mathcal{L}; \text{typecase } \tau e \rightarrow \mathcal{L}'; \text{typecase } \tau e \quad \text{ev:typecase-con} \\
&\mathcal{L}; e_1 \rightarrow \mathcal{L}'; e'_1 \\
&\mathcal{L}; e_1 [\tau] \rightarrow \mathcal{L}'; e'_1 [\tau] \quad \text{ev:tapp-con} \\
&\mathcal{L}; e_1 [\ell] \rightarrow \mathcal{L}'; e'_1 [\ell] \quad \text{ev:lapp-con} \\
&\mathcal{L}; e_1 [C] \rightarrow \mathcal{L}'; e'_1 [C] \quad \text{ev:sapp-con} \\
&\mathcal{L}; e_1 \rightarrow \mathcal{L}'; e'_1 \\
&\mathcal{L}; e_1 [K] \rightarrow \mathcal{L}'; e'_1 [K] \quad \text{ev:kapp-con}
\end{align*}
\]