Journey to the Ivory Tower: Making Links' Polymorphism Ad Hoc

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Henry Wandover

Annandale-on-Hudson, New York
May, 2025

Abstract

This paper outlines the philosophy and general rationale behind multiple contributions to the Links programming language, a language that attempts to ease the tribulations of designing modern web applications via the removal of "tiers." Both improvements tackle ad hoc polymorphism, one in an ad hoc way and the other through the addition of *subkind classes* to the Links type system.

Contents

Dedication

Dedicated to my mother, a woman of dedication and grace.

Acknowledgments

This paper couldn't have been finished without the invaluable guidance provided by Professor Bob McGrail.

1
Introduction

This chapter presents the Links programming language and also serves as a general summary of what is to come.

1.1 Why Links?

The software development industry as a whole has seen a general shift towards developing software for deployment on the Internet, the basis of which commonly relies on the well-known tools: HTML and JavaScript for the client, Python, PHP or Java for the server logic, and, of course (most commonly) some derivative of the SQL family for its database.

For the human programmer, this can be quite a nightmare. She is forced, by the current standard of web programming, to learn—and use—a multitude of tools to get her web application operational. Furthermore, this juggling of different languages does not make clear whether data coming from the front-end, or a database query, will correctly link with the back-end logic. This problem is known as *object-relational impedance mismatch*. This is most simply understood in the above stated example: data being queried is stored in relational data stores and the logic is a domain-driven object model [11].

This is the primary goal of Links [6], elimination of the impedance mismatch problem by providing a single implementation language for *all three tiers*. Some code translates to JavaScript, some translates to SQL, and what is left is interpreted directly by links for the server portion.

Because it is a single language, all written code is type-checked by its modified Hindley-Milner type system. The language itself is functional, typed, and strict. An important feature of the language design is types, which in practice means that data is consistent from HTML forms, to server logic, and database queries.

These design principles make Links a compelling solution to the challenges of modern web development. By unifying the three tiers of web programming into a single language, Links reduces the cognitive overhead for developers and mitigates the risks of mismatched data representations. However, there remains significant room for further enhancement of its type system and usability. The goal of this paper—and its accompanying code—is to contribute to the ongoing effort by enhancing the language's type system. The two alternate additions made to Links involve different ways of accomplishing ad hoc polymorphism, a concept that many should already be familiar with, but if not, its importance and conveniences will be elaborated on later. This paper will simultaneously attempt to serve as an introduction to links, type theory—with a special emphasis on kinds—as well as the implementation of ad hoc polymorphism.

1.2 Motivations

Currently, we are in a period of revival for programming language design. On the lower level, Rust and Zig are aiming to improve on C/C++ and on the web there are also a few contenders. One example is a functional programming language called Elm[7] that intends to improve the web application workflow with both a mature language and extension ecosystem. Elm's type system is, in practice, very simple. There is no subtyping (unions nor intersections), no flow-sensitive typing, first-order type constructors or a Hindley-Milner type inference, contrary to Scala, which has a much more complex type system that rivals Haskell. Or even TypeScript, which is limited by the fact that it has to work with JavaScript, leading many to want to refer to TypeScript as simply a 'linter'. However, all these examples have a different scope to that of Links. Interest in new web languages stem from the many failures associated with working with JavaScript. To name a few: no integer type (both integers and floating-points are of type

`Number`), all variables are global by default, both classes and inheritance are counterintuitive, and many operations commit strange sins (e.g., adding two arrays produces a string [33]). As Bjarne Stroustrup puts it: "There are only two kinds of languages: the ones people complain about and the ones nobody uses."

The name of this paper is a reference to a talk by Simon Peyton Jones regarding Haskell, speaking of its features as well as the complications of language designed by committee [13]. The inciting metaphor depicts the Haskell language as an "ivory tower," where its improvements come out of a desire to translate theory into practice. This comes at the cost of a perceived higher barrier of entry, as there is an extensive literature to be aware of—in either type theory proper, monads, effects or other subdisciplines—and there is not much interest in functional languages outside academia or open-source online collaboration.

Purposeful or not, on Links' site, its tagline states: "Links: Linking Theory to Practice for the Web." [5] However, this is not to say Links is useless as a language—or that it has no possibility, or future, of commercial viability. While it can be challenging in the early stages of use, as is any functional language when one is coming from an imperative-dominated world, it becomes clear there is a superiority here as compared to JavaScript. While still a fledgling language—still exhibiting years of extensive development—Links offers much more to the programmer as a functional programming language designed specifically for web applications, as well as a much more powerful typing system to boot. Furthermore, for the programmer hoping to implement new language features, Links operates similar to that of Haskell, where its ethos as a project allows itself to be treated like a laboratory. Where Haskell has more of a fascination with laziness, types and monads, Links finds itself, of course, more focused on the web domain, with interests that include row polymorphism, effect handling, concurrency, and a greater focus on kinds.

This is important to relay because for an audience outside of Edinburgh University, there is going to be an extremely small chance that anyone has encountered Links, interesting enough since there are notable names attached to the project, currently and in the past [12]. However,

this project is not so interested in a direct way of improving popularity but instead experimenting with ad hoc polymorphism. The project stems very casually from a tiny frustration with operators in Links, which has now eventually grown into a much larger project. In the most perfect of worlds, this introduction of user-extensible ad hoc polymorphism will encourage others to try Links with a much more common set of operators. It also continues this sibling relation between Haskell and Links, with members from the former community hacking on Links and many ideas being brought over.

1.3 Goals

There are a multitude of papers on the Links site, a good amount on effect handlers and others on database queries [24]. Yet, currently there lacks any work dedicated solely to Links' type/kind system. The goal of this project was to hack on Links but also provide a general introduction to what is going on behind the scene in Links' kind system. Along with that, there were numerous additions contributed to the language.

1.4 Contributions

The Links codebase was altered in numerous ways:

- Ad hoc polymorphism for arithmetic operators
- Dynamic kinding and subkinding
- A flexible subkind classes system
- Dynamic restriction resolution
- Dynamic constraint creation
- Expanding the Links Prelude with the `Num` and `Show` module

With these changes Links could ideally start to move in the direction of adopting the so-called *subkind classes* with this implementation and alteration of the system without breaking

the previous version. Much of the prerequisite work involved translating a static environment into a dynamic, safe, and efficient new representation.

1.5 Report Structure

The structure of this paper is as follows:

- Chapter 2, literature review;
- Chapter 3, type and kind theory (illustrated with Links);
- Chapter 4, the proper ad hoc implementation;
- Chapter 5, discussion of type classes;
- Chapter 6, the implementation of subkind classes; and
- Chapter 7, conclusion.

2
Literature Review

General information regarding the Links programming languages circulates around the homepage [5]. There are multiple published papers, both graduate and undergraduate, either projects made using Links, improvements to the language or the elder catalog that simply introduces and explains the language's purpose. Some of the genius of the Links language is that much of the early development and record shows that most of the emphasis was placed on getting the basis completed first. This served the dual role of "finishing" their project as well as building a community to help with further development. Both appear to have succeeded. Researchers and students have made working projects in the past with the language, and the change-log shows there have been progressive improvements made.

The goal of this paper is clear: to further develop and improve Links. Rather than starting from the ground up and building a toy language in which to create my ideal type system, it seems more worthwhile to contribute to a more mature project. This requires getting familiar with the syntax and use case of Links, the underlying implementation (which is written in OCaml), along with sourcing information about type theory. While all three considerations can be time-consuming in their own way, the information needed to do so is very easy to find. Mostly updated documentation for Links is tied to its webpage and OCaml has *Real World OCaml* [16] along with sufficient generated docs.

Considerations are still being made for the remainder of this project, as of the midways due date, but what is decided upon is more than substantial for serious considerations. As it stands currently, Links has no concept of type classes or ad hoc polymorphism. The arrival to this conclusion is shown in the documentation as well as issue #1070 [29], which laments about the necessity of -. when negating a float typed numerical. This syntactic notation exists for the other arithmetic operators as well. It would appear that this stems from OCaml's use of the same syntax. Three years ago, when this issue was posted, one of the developers agreed that it was an annoyance but didn't give a clear indication if it would be resolved.

By the nature of this project, the research is going to be multifaceted. Currently, there has been a strong focus on polymorphism as the method by which to solve this first problem. Both the *Dragon Book* [1] and *Elements of ML Programming* [28] present the basic idea of polymorphism sufficiently. The former is a more generalized description and has more of a bias towards C; however, it does illustrate the new challenges met by the type checker. And as will be referred to later, ML is distinct in that its polymorphic system is *closed*. Just as important is Wadler's article [30] on a "new" way to construct an ad hoc polymorphic system. This is of course through the use of type classes, which will be dissected further along.

Much of the knowledge of type inference and concepts such as unification (or substitution) was gathered from Thompson's book [27]. Alongside that, the code structure of Links, after getting used to it, very clearly constructs the story of a type checking system. While not directly referenced in any manual, following the flow of type variables in the internal representation made understanding more complex concepts like generalization easier, as it was less theoretical and more practical and algorithmic.

Based on communications with the Links team and the unresolved issue, something is yet to be done about this. While there are two other noted cases of projects about improving the current type system, neither were an attempt to absolve the lack of function overloading, needed in order to relax the unnecessary syntax for arithmetic operations. While this point of interest can boil down to philosophical differences, this feature should fit into Links quite comfortably.

As mentioned before, the lack of such a system is most likely due to interests in improving necessary infrastructure elsewhere. According to some thread on Links' board, it seems there is not much interest in working on the type system. However, there are still many complaints about the lack of a structured way of kinding [3] and using ad hoc operators.

3
Background

3.1 Type Theory

We are not going to get too deep into the sticky world of type theory, simply typed lambda calculus and type inferencing algorithms; however, it'd be useful to go over some things that pertain to the remainder of this paper, beginning with type theory. Type theory refers to the cross-divisional study situated between mathematics and theoretical computer science; type theory in *theory* refers to the formal presentation of an individual type system. What differentiates the type system is its rules and features. While it can be viewed as an academic study for functional programmers with too much grant money, type systems in practice are some of the strongest tools that a language can use.

For the uninitiated, it would be best to begin with types, then move onto kinds, and then fill in the remainder. Much of these concepts will be presented by proof as they normally are; only the important ones will be included in the paper proper, with supplementary items deferred to the appendix. Understanding the most simple and fundamental aspect of type theory is best put by Thompson: "Central to type theory is the duality between propositions and types, proofs and element." [27] This is illustrated as 'p is a proof of proposition P,' or

$$p : P$$

However, that is the logical way of interpreting that statement. For our purposes—and sanity—'p is a *member* of type P.' There are many papers and books on the logical system. For this paper, we will instead jump ahead with this idea of membership and types. Thompson's book is quintessential and provides superior explanations of judgments, proofs, and propositional calculus. The basic idea is that these same rules of mathematical proof can be used to also represent the computation of our computer programs, with the lambda calculus in that sense being the most simple programming language. This is in essence the Curry-Howard correspondence (or isomorphism as it is sometimes referred to) [21].

What then does our function Links say,

```
sig id : (a) -> a
fun id(x) {
    x
}
```

It's clear that all this function achieves is returning its argument. It is both polymorphic and universally quantified—two topics that will be touched on later. If we were to call `id(2)` then it would return `2 : Int` which again is computationally not that difficult or significant. However, without perhaps even recognizing it, we have constructed a logical proof, a proof of *implication*, specifically a proof of the implication $A \Rightarrow A$. If we were to write this by hand, it would appear as follows:

$$\frac{[x : A]}{\lambda x_A.x : (A \Rightarrow A)}[\text{Impl}]$$

Our top assumption is that $x : A$ allows us to deduce that x is of type A. Then, using a rule of implication or $\Rightarrow$, it is stated that the lambda expression has that type $A \Rightarrow A$ if the applied parameter is of type A. Simultaneously our function is defined as such type and a proof of the implication of $A \Rightarrow A$ is proven. This implication is expressed in links as the function type or `(a) -> a`. Again, this is a simple example, and things can quickly get more difficult and involved with several logical rules, but this is meant to illustrate why functional programming

is quite interesting. The focus on type systems' types and functions is a near-direct reflection of logical systems' propositions and proofs.

Hopefully, this point is already raising a few questions: "What if a type isn't known?" or "We have an idea of what a type is, but what the heck is a kind?" Maybe even, "How does polymorphism work?"—considering that it is in the title of the paper.

Again, while not the focus of this paper specifically, it should be noted that our example uses what is called the simply typed lambda calculus, as opposed to the slightly simpler untyped lambda calculus. The latter lacks types and therefore is far too general for our purposes. Furthermore, simply typed does not allow for type inference nor polymorphism, which is why one must look for a more robust system, such as found in what are aptly called type systems.

3.2 Type Systems

An implemented programming language uses a type checker, which is based on the syntax's information and notion of types along with the rules that determine how types are assigned to language constructs. According to Aho, Sethi and Ullman, there are kinds of types: basic or constructed[1]. The basic types are what any programmer is used to, in C they are integers, floating point numbers, characters and booleans (via `stdbool`). The constructed types are arrays, records and sets; by that logic, both pointers and functions can also be considered constructed types. As will be seen later, the usage of the word *kind* is intentional and not inconsiderate.

This idea of a type expression (or typed expression) is a *type* applied to what is called a *term*. For example, `2 : Int`. Or to connect back, 2 is a member of type `Int`; however, terms and types are more generally used.

The basic idea of type expressions is what informs the backbone of a type system. A type system provides a collection of rules for applying these expressions to multiple parts of a program. Pierce offers the following definition:

> A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.[23]

The type checking phase is where a type system's rules are defined and its whole is implemented. The power of type checking is understood in this distinction of static vs. dynamic. A *sufficient* type system—sufficient meaning it is performing an acceptable amount of work—sidesteps dynamic type checking for type errors by determining at compile time any type errors. Therefore, by statically checking for type, a sufficient type checker would assign a type other than `type error` to a part of a program. Thus, type errors cannot occur during the running of that part. One could furthermore say that a language is *strongly typed* if its type checker can guarantee that an accepted program will not encounter any runtime type errors.

It is important to note that not all type checking can be performed statically. Consider an example from Aho et al.

```
table: array[0 .. 255] of char;
i: integer
```

Then they say computing the `table[i]` shows the compiler cannot guarantee during a program's execution that the value of i will be within that range [1]. While list indexing is not possible, a similar problem can occur: Trying to take the head of a list will result in a runtime error. This is because it succeeds type checking, as the empty list is of type `[a]`; however, it was not confirmed during type checking if the list had any element to fetch. This is a limitation of the Hindley-Milner type system, which can effectively check for type; however, specific value is not necessarily accounted for as demonstrated.

3.2.1 The Hindley-Milner Type System

If one is familiar with any type system, there is a strong chance that it is the Hindley-Milner type system. One of many examples of double-barreled names, the type system was first described by the logician J. Roger Hindley and later rediscovered by computer scientist Robin Milner.

Coincidentally, Milner was working on his PhD thesis at Edinburgh University, the brain trust behind the Links language—which, of course, uses a modified Hindley-Milner.

One question from earlier that can be answered is how can a type be resolved? We say an expression E is opposed to a type T, or simply $E : T$. $E : T$? in this scenario would require *type-checking*, using context to confirm that the expression is of that type. If the expression is known only as $E : \_?$, and there exists some universe of types that can be pulled from to resolve this expression, then the algorithmic process to get a type for E would be type inference.

Hindley-Milner provides not only type inferencing and static typing, but also parametric polymorphism [19]. This is why it is still a popular choice today, used by Haskell, OCaml, Links, and earliest by ML [20]. However, it is not as complete as many would like it to be. And there are more solutions to the ad hoc polymorphic problem today than there have been in the past, but unlike parametric polymorphism, these solutions are often extensions of a type system. The '98 version of Haskell (Haskell98) introduced type classes [25] as an extension of its Hindley-Milner type system. On the other hand, OCaml neglects to add ad hoc at all, keeping their system pure. Before we get ahead of ourselves it would be useful to note the differences between these polymorphisms.

3.2.2 Polymorphism

Polymorphism is a concept, in both programming language theory and type theory, that can be simply translated to mean "one symbol represents multiple unique types." This is in contrast to *monomorphism*, where a symbol given any value can only be interpreted to inhabit only one type [2]. There are both polymorphic functions and types. The former is a function whose parameters can have more than one type, and the latter are types whose operations can be applied to values of more than one type. Polymorphism is most commonly understood by the average programmer, in the object-oriented paradigm as, say, Java's generics, as depicted in the following example:

```
static <T> void genericDisplay(T element) {
        System.out.println(element.getClass().getName()
                           + "=" + element);
}

public static void main(String[] args) {
    // Generic method with type Integer
    genericDisplay(11);
    // Generic method with type String
    genericDisplay("Hello, World");
}
```

This snippet defines a generic method (function) that serves the role of identifying the type of the method's argument. In Java, that unknown type is denoted as `T`. Upon running the code, the expected result is two lines:

```
java.lang.Integer = 11
java.lang.String = "Hello, World"
```

Strachey [26] is often noted for distinguishing between the two major types of polymorphism. *Parametric* polymorphism is implemented when a function operates consistently over a range of types that share a common structure, as is the case of the above generics examples. In Java, parametrized types work because they all share the role of being objects. Then, more importantly for this paper, there is *ad hoc* polymorphism, which is when a function *appears* to work on several unique types that often behave in different ways.

Furthermore, ad hoc polymorphism indicates that common polymorphic function definitions have a distinct definition for each type. This is in direct opposition to a parametric *uniform* operation, where the function acts identically and the type doesn't affect their role. A common example is **length** in Haskell, or really any language in the ML extended family, which returns the length of a list of any type. This means that the signature of length looks like:

$$[\alpha] \rightarrow Int$$

where α is any type or rather more professionally, a *type variable*. In this case, the datatype doesn't matter, the type variable maintains its generality since there is not a necessity to unify

biased by as specific constraint. So this function will work for lists of strings, integers, or even matrices.

```
length([3, 4, 5, 6]) == 4
length(["foo", "bar", "fun"] == 3
```

There has been work done in the past few years [5] to add first-class polymorphism, and then operation polymorphism, to Links. This extension gives the language and programmer the ability to treat polymorphic functions just like other values; they can be passed as arguments and stored in data structures. Pierce describes first-class polymorphism as the "most powerful form of parametric polymorphism." [23] However, what is yet to be implemented is ad hoc. The point can be made that ad hoc polymorphism will only be able to support a limited number of distinct types because a specified definition will have to be provided for each type [30]. However, this is where type classes play a vital role in the simplification and safety of the system. Of course, it is noted that this type of polymorphism is not a fundamental feature, but it can be argued that one goal of programming is to abstract when necessary and if it logically makes sense to do so.

To allow for our type system to properly check for the type of an expression, our type system must be extended to allow for type variables. This, of course, introduces problems that our simple lambda calculus is unable to solve. The utility of the Hindley-Milner type system is then to be able to properly infer the most general type in cases of application where type is not specifically denoted. When introducing subkind classes, there is a richer discussion of some of the rules found within Hindley-Milner that apply. It is easier to grasp the concept while working with the implementation of this system; however, I will present a simple explanation of what the type checking algorithm is doing.

On a high level, the type variables need to be properly formalized by mapping type expressions to said variables. This process is called *substitution*. Assume that a substitution is to be defined as S in some recursive function *sub*. We can encounter either some basic type, type variable or a function type, all represented by t, and the result of our substitution on a would be $S(t)$. The relationship between the application of substitution and t is that $S(t)$ is an *instance* of

t. However, when a substitution does not mention an expression for some variable α, then it is assumed that $S(\alpha)$ is α. This means that the substitution is the identity mapping on such a type variable. In Links, this relationship between substitution and expression can be demonstrated—using $\blacktriangleleft$ to demonstrate the instance is on the left.

$$\text{sig foo} : (Int) \to Int \blacktriangleleft \text{sig foo} : (a) \to a$$

$$\text{sig fadd} : (Float, Float) \to Float \blacktriangleleft \text{sig fadd} : (a, a) \to a$$

$$\text{Just(Int)} \blacktriangleleft \text{typename Maybe}(a) = [|\text{Nothing}|\text{Just} : a|]$$

$$a \blacktriangleleft b$$

This can fail if what appears on the left is not an instance of t on the right such as $integer \blacktriangleleft float$ which attempts to substitute a primitive type for another primitive type, or $Char \to a \blacktriangleleft a \to a$ which is an incomplete substitution.

With the ability to substitute type expressions comes the ability to *unify*. Links unifies both flexible and rigid variables. Restrictions are used in both but for flexible variables are checked directly to see if unification is possible. Given two type expressions t_1 and t_2, unification is only possible if $S(t_1) = S(t_2)$. According to Aho et al. [1] we are going to be interested in *the most general quantifier*. As previously discussed, a quantifier poses constraints on the instantiation of a type variable. This informs further the decision-making process of a subkinded type variable, which searches for the "substitution that imposes the fewest constraints on the variables in the expressions." What this actually looks like in Links will prove useful in a further discussion of this process.

Bounded Polymorphism

The main distinction lies in the differences between ad hoc and parametric; however, as noted in the introduction, subkind classes use the moniker of a *bounded* polymorphism. Wadler omits using this specific term, instead simplifying the point of type classes to be an "exten[sion] [of] the Hindley/Milner type system to include certain kinds of overloading, and thus bring together

the two sorts of polymorphism that Strachey separated." [30] Which is a sweet way of putting it, and perhaps at the risk of getting specific, some clarity might be lost, but either way, the use of bounded is meant to evoke a similar idea. Since this is functional programming and not C, the idea is to maintain a type safe system that catches any such type errors at runtime. However, we want the utility of the ad hoc way of life.

So that would involve morphing ad hoc to resemble more of what parametric looks like. One way to understand a type variable in Links such as `a` is to say it is universally quantified. However, as will be discussed with kinds, that universe is still limited to types within its higher order set. So if a parametric polymorphic function is "universal," then to implement an ad hoc polymorphism, that means shrinking the set of possible types from anything to only those that have implementations written. In a word, bounded. This same idea can be referred to as bounded quantification or constrained genercity as well [15].

This way, when the type system makes an attempt to instantiate some type variable with a type that has not been implemented, the type checker throws an error. What a quantifier is will be elaborated on further when discussing subkinds as well as the implementation of the subkind classes, which, like type classes, integrate this bounded form of polymorphism, borrowing concepts from both parametric and ad hoc.

3.3 Type Theory by Kinds

To understand kinds, one must consider their relationship to types.

3.3.1 Types v. Type Constructors

The difference between types and type constructors in terms of use is that a type can be used within your code on its own, while a type constructor requires additional parameters in order for it to produce a usable type. Furthermore, a type describes a datatype (e.g., primitive type, such as `Int`), while a type constructor describes a type family. Generic functions, in terms often

expressed in Java, are referred to as "parametric polymorphism" for this very reason; they are polymorphic over type parameters.

An example of this distinction is LinkedList(Array) as opposed to LinkedList(Array(Int)), the second case being one of a type constructor being employed. Functional languages with more developed type systems, such as Links, Haskell, or OCaml, find this distinction between types and their constructors expressed through the use of "kinds." At the risk of simplification, a kind can be understood as the "type of a type." The following are some common kinds and their definitions—along with examples.

3.3.2 Kind Definitions

`*` is the kind of every type. In Links' case, these are all base types: (within the subkind Base) Int, Float, Char, DateTime, String [1]. Links colloquially refers to this kind `*` as `Type` since along with base types, it also covers abstract data types like lists (e.g., [Int]) or constructed binary trees (e.g., BinaryTree(2), which are binary trees composed of integers based on the defined root).

`* -> *` is the kind of the unary type constructor, which takes a type as a parameter. In common tongue, this would be read as: "Given `*` implies a result of `*`." From the previous example, binary tree alone is of kind `* -> *` as it requires a type, as is the case with Lists and Tuples. Records in Links operate similarly but fall under the kind `Row`. Constructs of kind `* -> *` are sometimes referred to as "unary type constructors" because they require a single type to instantiate the next type.

`* -> * -> *` is the kind of a type constructor that takes two types as parameters, thus "binary type constructors." The most common example would be a structure like Map—not to be confused with the function often referred to as map. Most should be familiar with the concept of Map; but if not, it would work as such: defining a new Map of Map(String, Int), where given a string, it would return the respective integer value. In its initial construction, it

[1] In a previous version, String was an alias for [Char], or an array of characters. However, due to its backend exchange of data between Links code and the database, and how strings are represented in Links, this was changed [4].

produces * from its two type parameters. This can then go on: * -> * -> * -> * which takes its respective 3-type arguments.

Because the kind * would instantiate to a data type, kinds of * -> * and higher (type constructors) can be understood as "higher-kinded types," which is often the terminology used by those in the Haskell community.

A final note, often in languages that are inspired by Haskell, such as Links, one can expect to encounter what are commonly referred to as function signatures. In Haskell they would appear as:

```
add :: Integer -> Integer -> Integer
```

This states that a function add will take two integers and return an integer. Functions type and those higher-kinded types that go beyond unary operations use a technique called currying, which allows a single function with multiple inputs to be transformed into, in this case, two functions that each take one input [26]. Of course, this would not be done for our unary kind operations because those type constructors only ever take a single type as their arguments. To conclude, this function `add` would have the kind of * -> * -> * but once applied, its resulting kind would be * and instantiated type `Integer` [25]. In Links, these signatures are optional, as the type inference is better suited for its reduced set of available types. Regardless, I will be using signatures going forward. This addition function would appear as such in Links:

```
sig add : (Int, Int) -> Int
fun add (x, y) {
    x + y
}
```

3.4 Subkinds

> "Universal propositions, therefore, respecting the properties of superior Kinds, unless grounded on proved or presumed connection by causation, ought not to be hazarded except after separately examining every known sub-kind included in the larger Kind."

John Stuart Mill's *A System of Logic* [18] seems to be the very first reference, conceptually, of the term subkind. For him, kinds help group propositions, and subkinds, a way to further

distinguish between said kinds of propositions, are in unison able to build up a proof for the proposition. In a very abstracted sense, our concerns with subkinds are much the same, using a part of a larger (whole) collection to prove our function (or proposition) correct.

A definition more grounded in modern computing would be: A Subkind is a collection that represents *rigid* specifiers derived from a subset of its parent [kind or subkind]. By default, its usage does not require a relational dependency.

Rigid is emphasized because the types instantiated from a subkind will always maintain their "rigidity" following instantiation. That is, when a function is applied, the type variable can be instantiated to any type. Links describes its rigid variables as "proper polymorphic object-level type variable(s)," as opposed to *flexible* type variables. Perhaps slightly confusingly, a subkinded type variable can be denoted as a flexible variable in Links and the operator can still be used as a polymorphic operator, *if* there is a single application.

Flexible or Rigid?

This conclusion is based off the documentation, and in the implementation, it is clear that subkinds are utilized in resolution of flexible variables. The conclusion drawn is that in the absence of specific function signatures, a flexible variable is utilized. This can be exemplified by the following example:

```
fun f(x) { ignore(f("a")); ignore(f(1)); 1};
```

The recursive function `f` is rejected by the type error stating that an integer was applied to a function of type `(String) ~a~> b`. This indicates that the type checker resolved `f` to a monomorphic function and then moved on to apply integer and failed because it then received the wrong type. This can be resolved by adding a signature: `sig f : (a) ~> Int`, stating that the function `f` takes a rigid type variable. Let's say we want to visualize what the inference system is doing and use the signature: `sig f : (%a) ~> Int`. We will be met with the same error from before because the function is now monomorphic. This is to say that subkinds are rigid, but how Links works means that a flexible variable can be subkinded and still have any of the supported types allowed. Consequently, this is not a "proper" polymorphic type variable, as

exemplified by this example using recursion. When it is flexible, or lacking a signature, then after the first application, a type for the function is inferred and following uses will not be inferred again. Using the signature: `sig f : (%a::Base) ~> Int` rightly performs no differently.

The conclusion is that while subkinds can be used with flexible type variables, after the first application in an application, a function with a flexible variable loses its ability to be polymorphic.

Relationships

Furthermore, a subkind does not require a relational dependency; a subkind needs no "child" or "sibling" subkind. In our implementation, a parent kind or subkind is optional. Of course, to actually use a subkind, a kind must be determined during type checking. This is to allow for flexibility in usage of subkinds. Already present in the Links' system, the subkinds `Any` and `Base` are allowed to be subkinds of `Type` *and* `Row`. This is to enforce constraint on the types that can be instantiated to type and row variables, respectively. Although there are cases such as the subkind `Eff` which only has `Row` as its "super-kind," since in Links, function effects are represented by rows.[2]

However, once a type variable is properly instantiated and a primary kind is discovered, then a subkind will in effect have some primary kind (i.e., `Row`) as its ancestor. This allows for subkinding from a more general kind to the proper subset.

3.4.1 Boundedness / Quantifiers

Subkind is an esoteric term that rarely comes up in the programming community. For the working programmer, it can be tossed aside without regret. However, subkinds and the operationalization of said collection constructors is what this paper is all about. Very rarely—or if ever—have subkinds been employed in this schema to create what this section will refer to as *bounded polymorphism*. These are still ad hoc operations, but this idea of boundedness makes this approach very compelling.

[2]This was the case previously, but with the more structured subkind system now, it is clearly denoted that Eff is a subkind of Row alone.

Boundedness most directly refers to quantification, a subject of high concern for Links and its type system. Subkinding, if nothing else, is generalized quantification of kinds. At first glance, quantifiers can feel a bit obtuse, and when looking through the Links codebase, there are constant references to both quantifiers and quantification. Let's start with a simple example. In English, there are words that are so called "quantifier expressions," such as: "all," "some," "each," or "few." We could then combine our quantifier with a single (or plural) noun to create "some dogs" or "each question." Now we have a restricted quantifier phrase, i.e., not all the dogs, only some. Finally, we can take our quantifier phrase and apply it to a predicate and create a sentence: "some dogs are cute."

This is very a simple example, but it highlights the fundamentals: A quantifier is a kind of operator that specifies, or constrains, the individual units that can satisfy an open formula—a formula that contains at least one free variable. These "units" are derived from the "universe." In logic, that "universe" could be referred to as the domain of discourse or the universal set. For our concerns, in computer science, it is most often called the universe of types. That universe is populated by individual types, and, furthermore, there is a universe of kinds.

One quantifier that everyone should be familiar with is the universal quantifier, or $\forall$. Revisiting our `id` function in Links demonstrates taking advantage of the widest possible quantification.

```
sig id : (a) -> a
fun id(x) {
    x
}
```

This is a quite boring function that has no real use, but it does serve well as a first example of quantification. Ignoring the `sig` or function signature for now, the function takes in a variable `x` and returns it, simple enough. The next logical question would be: *what is* `x`? This is where our function signature comes in to communicate the function's contract. It says that it will accept a type variable `a` and return `a`. In the absence of kind/subkind denotation, Links assumes that `a` is of kind `Type` and therefore will be satisfied by any datatype within the universe of `*`[3].

[3]Unlike Haskell, in most cases, function signatures can be completely forgone. Links is simpler language, and therefore its type checker is better able to quickly check types and confirm type safety. For more complicated

Just like with the universal quantifier, one would say that this means "for all;" however, there also exists a `forall` keyword in Links that refers to a similar idea.

```
sig effectual : forall e::Row, a . (a) ~e~> a
```

Here its usage states that all occurrences of type variables `e` are of the kind `Row` and `a` of the kind `Type`. Moreover, Links performs this operation as well; if a type variable is rigid and affixed with a kind, then all the times that variables occurs following are of that quantification.

```
sig apply : (a::Numeric, ((a, a) -> a) ) -> a
```

functions, it is advisable to indicate signatures for both the reader and the interpreter, although, even with the new ad hoc system in place, Links is able to effectively discover the kind of best fit.

4
A Proper Ad Hoc Implementation

The first proper code section will highlight the first implementation of operation overloading in the Links language. It is lovingly referred to as a *proper* ad hoc solution due to its nature as an encoded solution that resists any sort of extensibility by the user. We do of course still make use of subkinds for our ability to constrain the allowed types. All code for this branch of the Links codebase can be found on my GitHub page [31].

This solution falls more into the tradition of standard ML [22], specifically the ad hoc ability of its equality operator. It should not require much convincing to establish that `==` is an ad hoc operator; both `3 == 3` and `"Hello" == "Hello"` will equate to true, but how that is determined in the implementation language requires different functions. The former integer equality would point to some low-level system call, while the string comparison could involve a looped comparison of each of its character's ASCII value. For type constructors, like a tuple or record, this equality type could also be applied as long as type within the constructor has the equality type. For example, `(int * string)` would be accepted by ML where `{i : int, r : real}` would not due to its usage of a real—not an equality type.

We are working with this principle, however; instead of subtyping, we are worried about subkinds. The subkind of a type variable determines if an instance has access to this operator, constraining the instantiation of the type variables.

First, we can take a look at how kinds and subkinds are parsed. In instances of type aliases, Links interprets a variable `a` as a `SugarQuantifier` which will be resolved in the desugaring of type variables. For example, we can create the following alias and an example function that uses it:

```
typename Maybe (a) = [|Nothing | Just:a|];

sig pop : ([a]) -> Maybe(a)
fun pop (l) {
    switch (l) {
        case [] -> Nothing
        case hd::tl -> hd
    }
}
```

This defines the type alias for `Maybe` which works as you would expect. A type variable of this type will either return `Nothing` or `Just` if it actually has a value. I have provided an example function `pop` which will perform a safer `hd` function on a list. Given `[]` it returns `Nothing`, while `[1, 2, 3]` gives back `Just(1)`. In the `typename` call, `a` is a type variable that makes this alias parametrized—just as a list constructor is a parametrized constructor, or `* -> *`. When encountering type variable, Links will use the following pattern matching.

```
let kind_of p =
  function
  (* primary kind abbreviation   *)
  | "Type"      -> (Some pk_type, None)
  (* either a value row or an effect row *)
  | "Row"       -> (Some pk_row, None)
  | "Presence" -> (Some pk_presence, None)
  (* subkind of type abbreviations *)
  | "Any"       -> (Some pk_type, Some (lin_any, res_any))
  | "Base"      -> (Some pk_type, Some (lin_unl, res_base))
  | "Numeric" ->  (Some pk_type, Some (lin_unl, res_numeric))
  | "Session"  -> (Some pk_type, Some (lin_any, res_session))
  | "Eff"       ->
    (Some pk_row , Some (default_effect_lin , res_effect))
  | k           -> raise
    (ConcreteSyntaxError (pos p, "Invalid kind:" ^ k))
```

Using either `::Row` or `::Any` has no real distinction in this system, as all kinds and subkinds are resolved at the parsing stage. An unmarked type variable, as in `Maybe`, is assumed to be of

kind `Type`. On the other hand, for data type's (i.e., a `sig`; `(a::Base) -> a`), there is a second pattern matching that will be used specifically for subkinds.

```
let subkind_of p =
  function
  (* subkind abbreviations *)
  | "Any"     -> Some (lin_any, res_any)
  | "Lin"     -> Some (lin_unl, res_any) (* for linear effect vars *)
  | "Base"    -> Some (lin_unl, res_base)
  | "Numeric" -> Some (lin_unl, res_numeric)
  | "Session" -> Some (lin_any, res_session)
  | "Eff"     -> Some (default_effect_lin, res_effect)
  | sk        ->
    raise (ConcreteSyntaxError (pos p, "Invalid subkind:" ^ sk))
```

This is because type variables in signatures have specific notation: `a` for a type variable, —afor a row variable and `{:a}`, or `{-}` for presence variables. For the most part, we will focus on type variables, as they are more intuitive and general purpose. Instead of being read as quantifiers, these kinded variables are data types, which have their own desugaring pass that converts them from their syntactic representation to semantic ones. This is necessary in order to properly type check said data types following all desugaring passes. The only changes made here can be seen. I added for both quantifiers and data types the ability to use our new subkind `Numeric`. It has both its own restriction and `lin_unl`. The latter is one of two parts that make a subkind—linearity and restriction—and in this case, Numeric defines its linearity as non-linear. This—in extremely simple terms—means that an input of the subkind has no proportional relationship with an output of the same subkind. If it were linear, in Links this linearity is represented by `lin_any`, then you can have an input x and y and when you, for example, double x then you can expect your y to be proportionally twice as large. This is not so important for our consideration, but we can confidently say that for members of `Numeric` no such rule applies.

How Links determines the deepness of a subkind is by reference to its restriction, which in this implementation is just, again, a pattern matching parameter. Specifically we use this function:

```
let min l r =
  match l, r with
  | Any, Any              -> Some Any
  | Mono, Mono            -> Some Mono
  | Session, Session      -> Some Session
  | Effect, Effect        -> Some Effect
  | Base, Base            -> Some Base
  | Numeric, Numeric      -> Some Numeric
  | x, Any | Any, x  -> Some x (* Any will narrow to anything. *)
  | Numeric, Mono | Mono, Numeric ->
      Some Numeric (* Mono can narrow to Numeric. *)
  | Base, Mono | Mono, Base ->
      Some Base (* Mono can narrow to Base. *)
  | Session, Mono | Mono, Session -> Some Session
  | _ -> None
```

We are only concerned with minimizing restriction in two places. First, when attempting to unify two flexible type variables, we confirm that both the linearity match and then perform a minification of both variables restrictions. Flexible type variables—also known as a *weak* type variable or *unification* type variable—are monomorphic meta-level variables that are instantiated to object-level types by Links type inference algorithm.

Ignoring void pointers, the C language demonstrates a clear monomorphic type system where all constants, variables, parameters and function results have a single unique type. This makes type checking easier but as expected is directly opposed to polymorphism. However, we must still account for flexible variables. In fact the user can create a flexible type variable in a data type signature by prefixing with a %, as in `"a" : (%a)`. This asserts the *shape* of an expression but offloads confirming the type to the inference algorithm.

If no minimum is found in the case of flexible variables, then an error is reported since the user is attempting to unify a type variable with no relationship. While not explicit in this instance of the Links subkind system, there exists a hierarchy of subkinds with `Any` and `Mono` sitting at the top. All restrictions when paired with itself will return itself, while `Any` will be minified by any other restriction.

The second place where `min` is used is in creation of constraint predicates. A constraint is a module type that is created alongside a restriction for a subkind in order to ensure a type or row can satisfy a subkind.

```
module type Constraint = sig
  val type_satisfies : typ -> bool
  val row_satisfies : row -> bool

  val can_type_be : typ -> bool
  val can_row_be : row -> bool

  val make_type : typ -> unit
  val make_row : row -> unit
end
```

It is quite simple, yet fundamental in catching type errors for our ad hoc operations but more generally any user misuse of a subkind. In terms of polymorphic concerns, we focus specifically on the satisfy functions that can be used to catch typ errors in application. For example, let's say we want to add two strings together:

```
<stdin>:1: Type error: The infix operator
    `+'
has type
    `(a::Numeric, a::Numeric) -b-> a::Numeric'
while the arguments passed to it have types
    `String'
and
    `String'
and the currently allowed effects are
    `wild:()'
In expression: "hello" + "world".
```

This is rightfully unacceptable because our operator only has instances for integers and floating point numbers. Furthermore, this maintains a semblance of type safety, as this error is caught during compilation or more specifically the type checking phase rather than runtime. The `can_type_be` and `make_type` functions are employed in the cases of flexible variables.

Compared with the `Base` constraint, the constraint for `Numeric` will be even simpler:

```
module Numeric : Constraint = struct
  open Restriction
  open Primitive

  module NumericPredicate = struct
    class klass = object
      inherit type_predicate as super

      method! type_satisfies vars = function
        | Not_typed -> assert false
        | Var _ | Recursive _ | Closed ->
          failwith ("[3] freestanding Var / Recursive /
----------Closed not implemented yet (must be inside Meta)")
        | Alias _ as t -> super#type_satisfies vars t
        | (Application _ | RecursiveApplication _) -> false
        | Meta _ as t -> super#type_satisfies vars t
        | Primitive (Int | Float) -> true
        | Primitive _ -> false
        | (Function _ | Lolli _ | Record _ |
           Variant _ | Table _ | Lens _ | ForAll (_::_, _)) -> false
        | ForAll ([], t) -> super#type_satisfies vars t
        | Row _ as t -> super#type_satisfies vars t
        | Absent -> true
        | Present _ as t -> super#type_satisfies vars t
        | Input _ | Output _ | Select _ |
           Choice _ | Dual _ | End -> false
    end
  end
  let type_satisfies, row_satisfies =
   make_restriction_predicate (module NumericPredicate) Numeric false
  let can_type_be, can_row_be =
    make_restriction_predicate (module NumericPredicate) Numeric true
  let make_type, make_row = make_restriction_transform Numeric
end
```

The constraint `Numeric` will only be satisfied by the expected primitive types `Int` and `Float`. We reject non-typed variables, other primitives, and type constructors such as records. While freestanding variables are not allowed, `Numeric` can still satisfy a function that uses a flexible type in its signature along with a `forall` variable denoted with the subkind. Finally, some type aliases are accepted but only if they are quite simple ones such as `typename Nat = Int;`.

In the `TypeSugar` module, the different constructs of the language make their calls to other modules that handle desugaring or resolving. For this implementation, there are not any nitty-

gritty changes. All that has been updated are two utility functions that return the type of unary and binary operators. For unary operators, their signatures have changed from `Int` to `Numeric`:

```
...

| UnaryOp.Minus      ->
    add_empty_usages (datatype "(Int) {|_::Any}-> Int")
| UnaryOp.FloatMinus ->
    add_empty_usages (datatype "(Float) {|_::Any}-> Float")
| UnaryOp.Minus      ->
    add_empty_usages (datatype "(a::Numeric) -> a")
| UnaryOp.FloatMinus ->
    add_empty_usages (datatype "(Float) -> Float")

...
```

The `-.` operator's signature has been preserved so that previous Links code is not broken by the introduction of the ad hoc operator. Because of the heavy usage of arrows and symbols that contain hyphens, the minus operators have been specified. Similarly for specific binary operators the signatures have been included:

```
...

| Name "+"
| Name "*"
| Name "/"
| Name "^"      -> add_empty_usages (datatype "(a::Numeric, a) -> a")

...
```

Their final alteration to this branch was in the Lib module, which is a file that contains part of the prelude that can't be properly bootstrapped, as it relies on lower-level functions or other functions where it is easier to just write thin wrappers, such as `length` or `hd`, that are borrowed from OCaml.

```
  "+",          numeric_op ( + ) ( +. ) PURE PURE;
  "-",          numeric_op ( - ) ( -. ) PURE PURE;
  "*",          numeric_op ( * ) ( *. ) PURE PURE;
  "/",          numeric_op ( / ) ( /. ) IMPURE PURE;
  "^",          numeric_op (pow) ( ** ) PURE PURE;
  "mod",        int_op     (mod)   IMPURE;

  (* For backwards compatability *)
  "+.",         float_op   ( +. )  PURE;
  "-.",         float_op   ( -. )  PURE;
  "*.",         float_op   ( *.)   PURE;
  "/.",         float_op   ( /. )  PURE;
  "^.",         float_op   ( ** )  PURE;
  "^^",         string_op  ( ^ )   PURE;

  "abs",
  (p1 (fun n -> match n with
  | `Int _ ->
    Value.box_int ( let x = (Value.unbox_int n) in
        if x > 0 then x else -x )
  | `Float _ ->
    Value.box_float ( let x = (Value.unbox_float n) in
        if x > 0.0 then x else -.x )
  | _ -> raise
    (runtime_type_error ("Cannot compute absolute value:"
        ^ Value.string_of_value n))),
  datatype "(a::Numeric)->a",
  PURE);
```

The signature of the five operators are now `(a::Numeric, a::Numeric) -> a::Numeric` but the actual implementation has to be updated. As seen, they now use a new function `numeric_op` in place of `int_op`, which performs a runtime pattern match. As noted by Wadler and Blott [30] if the type of a function references the new subkind, then one can't expect to resolve such an occurrence until runtime, such as in the case of a function `square`:

```
sig square : (a::Numeric) -> a::Numeric
fun square (x) {
    x * x
}
```

However, for this implementation, there is no need to keep a dictionary of overloaded functions because the numeric functions are defined in Lib. Additionally, there is no sort of translation being done with ad hoc functions. Since I am using subkinds, its constraint will restrict the

types that can be applied at compile time, and during runtime application, there is no reason why the function should fail. I have included a runtime error call; however, this should never be used since during type checking, we have confirmed that such a function `square` only accepts integers and floating points. There are as well errors for arity and purity that I have added, as other Lib functions have these worst-case checks.

```
type pure = PURE | IMPURE | F2 of (Value.t -> Value.t -> pure)

let numeric_op impli implf purei puref :
    located_primitive * Types.datatype * pure =
  ('PFun (fun _ args -> match args with
  | [x; y] ->
        (match (x,y) with
          | ('Int _, 'Int _) ->
            Lwt.return ('Int (impli (Value.unbox_int x)
                (Value.unbox_int y)))
          | ('Float _, 'Float _) ->
            Lwt.return ('Float (implf (Value.unbox_float x)
                (Value.unbox_float y)))
        | _ -> raise
            (runtime_type_error "type error in numeric operation"))
  | _ -> raise (internal_error "arity error in numeric operation"))),
  datatype "(a::Numeric, a) -> a",
  F2 (fun l r -> match (l, r) with
          | ('Int _, 'Int _) -> purei
          | ('Float _, 'Float _) -> puref
          | _ -> raise (internal_error
            ("cannot establish purity in numeric operations"))
  )
```

This ad hoc implementation is quite simple compared with the latter, more substantial, introduction of subkind classes. However, I believe this still warrants its own section and explanation because it provides both a slick and safe way of achieving a subset of our goal, as well as is currently under consideration for a pull request. It also helps ease us into the more involved bounded scheme of polymorphism, which of course is still ad hoc in nature but provides certain checks in the expanded type system. Depending on the goals of the language, this ad hoc solution should suffice as it did for Starchey's ML. But we are in the post-modern era of functional programming languages, and there is no reason to stop here and search for alternatives.

5
What's a Type Class?

With some code written and considerations taken in implementing the previous ad hoc solution, one can easily see the limits of this deal. Not only does it not offer the user-extensible environment one comes to expect in functional programming, but the implementation requires a lot of fiddling to get set up. Links is a relatively large codebase, and in adding just these handful of overloads, we traversed many places that would make it a headache to continue adding ad hoc operations. Nor would it be possible to include new user-defined types. Take the following as an example:

```
typename Nat = [| Zero | Succ:Nat |];

sig addNat : (Nat, Nat) -> Nat
fun addNat(n1, n2) {
    switch (n1) {
        case Zero -> n2
        case Succ(pred_n) -> addNat(pred_n, Succ(n2))
    }
}
```

In Links we have defined a new type alias `Nat`, which stands for natural numbers. This could prove useful when one wants a type that is ensured not to be negative without needing a conditional to check an integer-typed value. That is apparent in its definition; it can only be `Zero` or a successor of `Zero`. Furthermore, this could easily be then converted to a value of type `Int`—if need be.

While this could be used as is, it is less than desirable for the same reason that using `.+` for floating-point addition is not ideal. One comes to expect that `+` should be used for all operations related to numeric values. As has been shown, it is possible to use subkinds in Links to achieve ad hoc polymorphic behavior. But unless the type `Nat` was added as a new primitive or default alias, it would be impossible to implement. This roadblock can be resolved, and in the current standing, *type classes* still are the best way forward. However, what is implemented into Links differs slightly as it elaborates on the current usage of subkinding in Links.

When Philip Wadler and Stephen Blott introduced type classes to the world, there was a sudden gathering of all functional programmers in the center of Jerusalem for three days of incessant prayer. In reality, what Wadler and Blott had formulated revealed to the functional programming world that it was possible to have a type safe (Hindley-Milner extended) system of ad hoc polymorphism. And this was done in what Wadler called a *less* ad hoc way. This is a nod to how exactly type classes are able to gift this ability to overload functions. It is also the fundamental difference between ad hoc and parametric, that is, the quantification or what set of types some functor is working with. In the case of parametric polymorphism, those types have at least one type parameter, which is later instantiated to any type. If we look to the higher order, that is thus the kind `* -> *`; this means in simple terms that the instantiated type doesn't matter—in this particular operation. Let's look at an example in Links (where the signatures are included to make clearer to the reader):

```
unsafe sig length : ([a]) -> Int
fun length (l) {
    unsafe sig aux : ([a], Int) -> Int
    fun aux(l, n) {
        switch (l) {
                case []     -> n
                case _::l -> aux(l, n + 1)
        }
    }
    aux(l, 0)
}
```

As was relayed previously, the type variable `a` does not matter in this case, as we are inquiring about the the type `a`'s construct—the list. Whatever `a` ends up being instantiated to is not

of concern to this function. In more specific terms, this length function uses a generalized type variable that is instantiated and thereby resolved upon application to a list of, say, type `Char`. It's important to acknowledge the difference between types of polymorphism here, as they operate very differently in practice. This could also be expressed as follows:

$$length : \forall\alpha.List\ \ \alpha \rightarrow \mathbb{N}$$

5.1 Type Classes

While type classes should be mentioned, this paper's subkind classes differ slightly in important ways, those of which will be covered in the following section.

Type classes are implemented, in Haskell at least, in a system known as *dictionary passing*. While this method is historically the second, it still remains the most popular method.

Just a year prior to the Wadler paper, German computer scientist Stefan Kaes published a paper detailing the discovery of his own type system. That 14-page paper shares some similarities to Wadler's proposed system [14]. However, there are limitations to intensional type analysis that are resolved thanks to the deliberate grouping of operators into so-called classes. There are further improvements to note going further, and it is worthwhile to do so, as other functional programming languages such as Scala or alternative Haskell compilers (most notably JHC [17], and in the past Chameleon), use a variant system based on Kaes.

As a side note, intensional type analysis gets its ideas from intensional logic. In that way, it is an extension of first-order logic, where quantifiers range over individuals in the universe—again, in the first-order. These quantifiers range over the set of terms that may have such individuals as their value. Present in this logic is a distinction between extensions and intensions, the individuals of the universe being the former and the terms that have the possibility of their value being one of these individuals the latter. This is most readily of usage in Martin-Löf's dependent type theory and not necessarily useful in our conception of both type and subkind classes since we are working with a modified Hindley-Milner type system in Links.

Type classes can be thought of as a kind of bounded quantifier. At compile time, the class is transformed into a dictionary that contains all the signatures required of the class. Then each operation is given the respective key in the dictionary. Next, every instance is created as an abstract datatype and required to implement all the designated signatures; if that conditional is met, then the new datatype, say `numDInt`, is created. Now every function that makes use of notation that constrains a type variable to a certain class swaps that information out for the abstract datatype that contains the dictionary. It takes a type variable as its type, then constructs a type based on the context.

Functions that make use of a type class in Haskell are thereby resolved at runtime because the actual value being applied to a function is not known at the end of type checking. However, it is still confirmed that a type applied must be instanced by the type class at that point. The canonical example is `square` which can be type checked to only accept integers and floats. But at runtime a dispatch table properly sets the correct implementation for an overloaded operator [30].

5.2 Subkind Classes

Haskell's type classes do know of the concept constraint: `Num a =>`. This is declaring a constraint on the possible types that can be instantiated to the type variable `a`; in this case, `a` is constrained to numbers (integers, floating point, etc.). While similar conceptually, subkind classes aim to generalize this system by making constraint and restriction a part of the kind system. By piggybacking off the current subkinding algorithm, we can introduce new subkinds in our code and, in effect, extend the system. This requires, among other things, expanding constraints and restrictions to be more dynamic, unlike how they were originally represented in previous versions of Links.

This idea was spurred on by an email exchange with James Cheney while seeking assistance with getting in to Links while originally focusing specifically on the ML-like ad hoc operators. Because Links has more of a focus on subkinding and kinds as constraints on what data can go

where, the idea was born out of working with the system rather than re-implementing a solution from some other context. It is also not outrageous to consider having type classes for Links. Why nothing has been done is due to a lack of interest in that area. As for the people still working on Links, there is more interest in the capability of Links as it relates to database query [24] as well as effects related research [8].

6
An Implementation of Subkind Classes

This update to Links adds two new keywords: `class` and `instance` [32]. They operate in much the same was as they do in Haskell and other languages that implement type classes. Of course, what is happening in the system is slightly different, but syntax was preserved for ease of use. In terms of what is accepted during the parsing stage, an instance must be a block with implementations of a class's operators. However, technically, a class can be both a block as with type classes, as well as a declaration. A declaration does not define any operators, instead emulating the behavior of the old subkinds. Therefore just inserts a new subkind with no related functions into the environment.

Revisiting our numeric subkind, if we wanted to overload `+`, `*` and other operations, we can now create a subkind class. To maintain similarities with Haskell, we will call it `Num` and define its class declaration as such.

```
class Num : (a::Type(Unl, Any)) {
                sig + : (a, a) -> a;
                sig * : (a, a) -> a;
                sig - : (a, a) -> a;
                sig `negate` : (a) -> a;
                sig ^ : (a, a) -> a;
                sig `abs` : (a) -> a;
}
```

This declaration states "a type variable `a` can make use of the operators `+`, `*` and so on as long as it belongs to the subkind `Num`." At this point, the operators won't actually

do anything because the subkind's constraint upon creation is set to allow no types. If one was to call, say, `(^)`, it would be accepted and the return of the program would be: `(a::(Unl,Num), a::(Unl,Num)) -> a::(Unl,Num)`. The operator is properly overloaded, but to actually make use of it, an instance must be called. Provided are two instances for `Int` and `Float` types.

```
instance Num : Int {
                + : addInt;
                * : mulInt;
                - : subInt;
                `negate` : negInt;
                ^ : powInt;
                `abs`: fun(i) { if (i < 0) `negate` i else i };
}

instance Num : Float {
                + : addFloat;
                * : mulFloat;
                - : subFloat;
                `negate` : negFloat;
                ^ : powFloat;
                `abs`: fun(f) { if (f < 0.0) `negate` f else f };
}
```

Consider the instance of type `Int`, it states that "there are functions `+`, `*` and so on, defined by its type `Int`." How that is confirmed will be further elaborated on. The same logic applies to the instance of `Float`, and functions such as `addInt` are defined in Links' `Lib` module—as discussed in the previous implementation.

Additionally, because a subkind has been created, `Num` can be used in function signatures as well. Take the function `square` as an example [30]:

```
sig square : (a::Num) -> a
fun square(x) {
                x * x
}
```

The signature is still optional, as the current inference algorithm can properly infer the type of square. This ensures type safety as well as prevents encountering a runtime error related to improper type application. This function indicates that only types within subkind `Num` will be

accepted, as they have all been confirmed to have implemented `*`. For example, if the instance of `Float` did not define `*`, Links will fire off a new error `IncompleteSubkindInstance`.

As one would expect, this function can be applied to integers and floating point numbers. But trying to apply a string will cause an error at compile time that `String` is not a member of `Num`. This is caught due to the improvements made to constraints, allowing the dynamic creation of constraints.

6.1 Restrictions

The restriction module is very closely tied with subkinds, so much so that we are going to assume that every subkind will create a new restriction, as well as constraint, that all share the same name. Perhaps there is some use case that could benefit from the usage of a current restriction, but to ensure that the kind system is stable, we won't allow re-use of restrictions. Instead they should remain as they are, indicators of the ancestry of subkinds with a single restriction referring to a single subkind class or declaration. It will be shown later, but to preserve native subkinds as they were, there are two options for a subkind, either a declaration or a class, the class being our polymorphic constructor and a declaration that simply constrains types as the original purpose was of subkinds.

Initially, restrictions were a static pattern matching, most likely for all of Links' history. If one wants new subkinds, then they would have to hard-code them into the system; however, this is a headache. The "proper" ad hoc solution to function overloading takes a while to implement. One has to go all over the codebase to find the areas they want to update. Of course, with time, you can become more familiar; yet, many seem to lament the lack of a structured way of addressing the ad hoc problem. For a time in the early development phase of this project, restrictions were a "dynamic pattern matching" gorgon. This would mean that when a new subkind was introduced to the environment, then a new restriction was created and placed into a dynamic OCaml pattern matching. The idea was to attempt to maintain much of the structure of restrictions as they were initially.

This worked, but it was quite nasty code. So instead, I opted for a hash table that tracks restrictions and their ancestors. Rather than pattern matching every given case, I instead compressed some of our assumptions. Two restrictions will always reduce to themselves because there is no possible way to minify such a case. The single caveat is `Any`, which when paired will always return the other restriction since `Any` is the default restriction when no subkind is specified. However, let's say we want to include a subclass. Then the restriction system should account for other possible restriction rules that are not descendants of `Any`. Use of the restriction table, without the optional argument, will default to using `Any` and `Mono` but if specified will instead set its ancestors to a "super-subkind." This makes it so when using flexible variables for a function's signature, the inference algorithm will be able to properly identify that a type variable is of a certain subkind.

In cases of a subclass, it has access to its superclass's functions in a generalized way. Let's say we have classes `Eq` and `Ord`.

```
class Eq : (a::Type) {
        sig `eq` : (a, a) -> Bool;
}

instance Eq : Int {
    `eq` : eqInt;
}

class Ord : (a::Type(Unl, Eq)) {
        sig `lteq` : (a, a) -> Bool;
}

instance Ord : Int {
        `lteq` : fun(x, y) {
            if (x `eq` y || ltInt(x,y)) true
            else false
        };
}
```

The equality class is trivial, and the ordering class's use of `eq` is not a result of inheritance but due to the fact that the type checker can confirm that `x` and `y` are integers in its usage. This utilizes the dynamic restrictions, as the lambda function that utilizes `eq` lacks signature and is determined by inference.

In these condensed classes, one should notice that `Ord` relies on the ability to check for equality when prompted to see if two values are less than or equal to. This overloaded function `eq` will have instances that will be adopted into its subclass to then be used in its own instances. Because these classes are very much ingrained within the current subkind system, the type checking system can then already verify that `eq` has such a subkinded signature without translation. When creating instances of `Ord`, the constraints should have been widened for `Eq` and therefore when making use of its operators later will rightfully not raise any type error. Of course `Ord` will have its own associated constraint—which is touched on next—but what is important is that when the time comes to make instances for our ordering class, because it is a child of the equality class, it in a sense inherits its functions without requiring any direct process of passing `Eq` operators to `Ord`.

This makes use of Links' functions `eqInt`ins and `ltInt` that essentially limit the types of equality operations. This is because Links currently makes use of OCaml's equality operators that have their own ad hoc implementation respectively. However, this example does work in the Links implementation with subkind classes because of the ordering restriction being a descendant of `Eq`. [1]

To demonstrate this system of inheritance, we can create this function that makes use of both operators: `eq` and `lteq`.

```
sig compare_adjacent :
    ([a::Ord], (a::Ord, a::Ord) ~b~> Bool) ~b~> Bool
fun compare_adjacent(l, cmp) {
    switch(l) {
        case [] -> true
                                    case [_] -> true
        case x::y::rest -> cmp(x, y)
            && compare_adjacent(y::rest, cmp)
    }
}

fun all_equal(l) { compare_adjacent(l, (`eq`)) }
fun is_non_decreasing(l) { compare_adjacent(l, (`lteq`)) }
```

[1] I have chosen to use lteq and eq for operator names because the current system already has such operators in use as previously explained. Additionally, the current implementation lacks a clear way to overload an overloaded operator, meaning using a class to add further behavior to an operator is something that is yet to be added.

As indicated by the signature, this function requires a list and a function that takes two variables of subkind `Ord`. Then the next two functions, `all_equal` and `is_non_decreasing`, allow lists of type variables of subkind `Eq` and `Ord` respectively. The type checker accepts these functions, indicating that both the equality and less than operators work for a function that is designated of subkind `Ord`. THis means that it will have access to the subkinds functions and its super class's functions.

In absence of a signature, the `compare_adjacent` function assumes a general type variable of kind `Type`, which, of course, does not break anything, but now if another function was made utilizing that function, more types are allowed. It is constrained in this example to show how a subkind's super class can be used by a child. Another example of this usage can be found in the appendix.

6.2 Constraints

More verbosely expressed in the realm of *subtyping*, a constrained type is a simple type combined with a set of subtyping constraints on its type variables [10]. This is a more well-tread topic when discussing subtyping; however, in our subkinding system, we still need a sense of constraints. Of course, there are restrictions in Links, but those serve as simple keys, keys that point to constraints. Not all restrictions have constraints, but every constraint has restrictions. For example, the restriction `Any` has no associated constraint, as it can take any type, type alias or even function. While on the other hand, `Base` has an associated constraint of the same name (there is little point to having any onomastic difference), that constrains subkinded type variables to a set of possible types.

The implementation for dynamic constraints was done in a way that minimally disrupts previous code structure. This means that some areas are debatably less than ideal OCaml code. However, for the goal of accomplishing a type safe bounded polymorphic system, it achieves that. Now it is possible to catch misuse of ad hoc operators at the unification stage and before the translation into the IR. While it is possible to sidestep the need for pattern matching

restrictions, with constraints there is a greater necessity to rely on such an ability because it is not possible to check for equality between different sorts of types in the Links representation of types. This means that it is not that simple to see if a polymorphic variant is the same as a primitive type. So there continues to be a reliance on pattern matching. Of course, a grander redesign could accomplish a better solution; however, out referential pattern matching should suffice for now.

To reiterate why constraints are important, consider this snippet:

```
instance Ord : Float {
        `lteq` : fun(x, y) { if (x `eq` y || ltFloat(x,y))
        true else false};
}
```

Such an instance is illegal and reported by the type checker. Specifically, why this is wrong is because in our attempts to create a new instance of `Ord`, we ignored the fact that `Eq` had no instance of `Float`. This demonstrates the safety associated with super classes but also the general checking of allowed types, which might be a bit clearer if we were to attempt to use the `square` function from before on `true`. The type checker again would fire a type error because booleans are not an instance of the class `Num` and therefore are not a member of the constraint `Num`.

Translating constraints to a dynamic constructor involved mainly two modules `ConstraintEnv` and `DynamicConstraint`. The former performs as it sounds; it tracks what constraints that Links is aware of when attempting to unify a subkinded type variable with some type. It's functionality is limited to registering new constraints, finding a constraint, and updating some constraint. Attempting to create a new constraint will cause an internal error. This would occur if two classes used the same name, and therefore the same constraint, but that is caught earlier on when adding subkinds to the subkind environment. Constraints are tracked in a hash table so adding and accessing are speedy.

To preserve the current structure of constraints, instead of altering the constraint module, a new sort of constraint was added. A dynamic constraint tracks the applicable types in a set. Upon creation, that set is empty as well as creating an empty constraint:

```
let create name =
    let module Instance = struct
      let name = name
      module Predicate = struct
        class klass = object
          inherit type_predicate as super

          method! point_satisfies f vars point =
            match Unionfind.find point with
            | Recursive _ -> false
            | _ -> super#point_satisfies f vars point

          method! type_satisfies _vars = function
            | Not_typed -> assert false
            | Var _ | Recursive _ | Closed ->
              failwith
                ("[3] freestanding Var / Recursive /
-------------------Closed not implemented yet
-------------------(must be inside Meta)")
            | _ -> false
        end
      end
      let type_satisfies, row_satisfies =
        make_restriction_predicate (module Predicate) name false
      let can_type_be, can_row_be =
        make_restriction_predicate (module Predicate) name true
      let make_type, make_row = make_restriction_transform name
    end in
    ConstraintEnv.register name (module Instance : Constraint)
```

Similar to how static constraints are created, a predicate is made that inherits from a general predicate that traverses a whole type. Making use of that super predicate allows the use of more complicated constructs such as polymorphic variants. However, upon creation, the type satisfies to nothing except for strange cases where the type system fails to assign a type or using freestanding variables, which is not supported by any constraint in Links. Using that predicate as well as the name of the constraint, which is always the name of a restriction, the functions of a constraint are properly created using auxiliary functions that propagate the specifications of the constraint and the restriction that it is related to.

Updating a class's constraint is similar to the creation, as it reconstructs the predicate using an updated set of types. If it is unable to find that set, it assumes an empty set. Then it updates

that set with the argument to the function before finally replacing the old set. If a constraint is being updated, then it means it is no longer empty; therefore, the predicate is now actually checking types.

```
let update name typ =
 match ConstraintEnv.find name with
 | Some _ ->
   let existing = Hashtbl.find_opt type_patterns name
                  |> Option.value ~default:TypSet.empty
   in
   let known_typs = TypSet.add typ existing in
   Hashtbl.replace type_patterns name known_typs;

   let module Instance = struct
         ...
         method! type_satisfies vars = function
           ...
           | (Alias _ | Meta _) as t ->
               super#type_satisfies vars t
           | (Record _ | Variant _) as t ->
               super#type_satisfies vars t
           | ForAll ([], t) -> super#type_satisfies vars t
           | Row _ as t -> super#type_satisfies vars t
           | Absent -> true
           | Present t ->
               if TypSet.mem t known_typs then
                   super#type_satisfies vars t else false
           | t -> TypSet.mem t known_typs
     end
   end
   ...
   end in
   ConstraintEnv.register ~replace:true
       name (module Instance : Constraint)
 | None ->
   failwith "Attempting to update constraint that doesn't exist"
```

Utilizing the super type sanctification function, it is able to decipher things such as aliases. If `typename` was used like `typename DInt = Int;`, then it should be allowed to use operators that are instanced by `Int` because it is the same type behind the alias. A similar notion is assumed for records, variants, and statements of `forall`. It should follow that if the types that comprise a record or variant are instanced by a class's constraint, then that type constructor

can be operated on. The patterns `Absent` and `Present` refer to presence variables; a variant's member can either present or absent. Consider this example:

```
typename Temperature = [|Cel:Float|Far:Float|];

sig checkTempEquality : (Temperature, Temperature) -> Bool
fun checkTempEquality(x, y) {
    switch ((x,y)) {
            case (Cel(m), Cel(n)) -> m `eq` n
            case (Far(m), Far(n)) -> m `eq` n
            case (Cel(c), Far(f)) -> (1.8 * c + 32.0) `eq` f
            case (Far(f), Cel(c)) -> (1.8 * c + 32.0) `eq` f
    }
}

instance Eq : Temperature {
    `eq` : checkTempEquality;
}

Cel(10.0) `eq` Far(50.0)
```

When applying `eq` to the variant type `Cel` and `Far`, only one is present, while the other is absent—in this case since there are only two possible options. Additionally, the type checking will fail if an instance of `Float` is not provided for the equals operator because the `checkTempEquality` function relies on that. But even if we were to say replace `eq` with `==` operator—utilizing the OCaml definition—this time the instance will fail to type check because while attempting to substitute the generalized `eq`'s variables, it is discovered that the variant makes uses of a type that is not properly instanced. A check that ensures when a polymorphic variant is used there is always going to be an operator that can operate on one of its rows.

Hopefully, this discussion has illustrated why constraints are crucial to the ability of ad hoc polymorphism. Without them, operation overloading could still feasibly work but would lack a guarantee of type safety. Even in the case of a function, where a variables uses a subkind and thus is computed at runtime, such type errors are still caught via the type checking of an application to the function. This approach differs notably from Haskell's type classes, where instances populate a dictionary and dictionary-passing is used. In contrast, because Links employs subkinding, no such dictionary is necessary at the type-checking stage. This reflects a

broader generalization: subkind classes leverage the constraint module to restrict which types can be used with their operators. Rather than utilizing a dictionary, an overloaded operator is given a signature associated with a subkind, which can be incrementally widened to permit more instances. A dictionary is only introduced later, to dispatch the correct implementation during application, but the typing itself is governed by a subkind's constraints.

Overloading, An Overloaded Operators?

I considered the possibility of extending the range of an operator beyond the constraints of the class but sided against it in the end. Assuming that `+` has the signature `(a::Num, a) -> a` and imagine we wanted to extend the additive operator be used to add a new leaf to a tree structure. This would not make sense to be an instance of `Num` because it would have to adopt all the other operators so a new class would have to be created. Currently creating a new class with an already overloaded operator will rewrite the signature of the function, in this case to: `(a::(Unl,Tree), a::(Unl,Tree)) -> a::(Unl,Tree)`.

This isn't so much a bug than it is a choice. Perhaps it would be useful to throw an error that informs the user that the operator is already in use; however, as it stands, one can still gather as much. The point is that I could see the merits in overloading an overloaded operator, thereby creating a new restriction that is a parent of both classes. So the `+` operator could be used in more situations. The implementation wouldn't require that much more effort; however, I believe that this causes unnecessary complications in the reading of Links code. Perhaps this is a moot point, but it is merely being brought up to both give recognition to a decision that was made as well as offer the potential for further improvements.

6.3 Further Elaborations on the Implementation

Subkind classes are more than restrictions and constraints; however, it is good to mention them first as they play a significant role in both specification of the subkind as well as type checking its instances and usages, although how that is actually done has not been discussed.

This section and the following are significantly more technical, as they describe how things are implemented internally in Links. It is worth discussing since there are not many papers on this area of Links, and it can be quite overwhelming if one wanted to contribute to Links specifically. The goal then is to both in detail explain the subkind class implementation as well as provide a general framework in which to tackle programming a new extension to Links that involves datatypes, types, kinds or quantifiers. However, the previous sections do cover the general idea of subkind classes and provide ample examples of how they can be used. That being said, the early discussions of type theory and concepts such as generalization are very much explored in the remainder.

6.3.1 Sugar Representations

Subkind class declarations are represented in sugar similar to alien blocks. Alien blocks are simply containers of multiple alien (exterior JavaScript) function signatures to be used in Links. Unlike the alien blocks, it is impossible to make a single class function line for obvious reasons. Furthermore, all subkind class and instance-related constructs are associated with `bindingnodes`, which cover alien calls, functions, variable assignments, and importing of other modules (most often another file). As mentioned before, at the parsing stage, there is no chance to encounter a single function call, so a class is read in as `Class of Class.multi Class.t`. The guts of the `Class` module are not terribly interesting, but they are extremely useful. Essentially, a class is created with a name, a list of quantifiers, and a list of function signatures. All of these parameters are required except for the signature list because an empty list is possible, and that could happen in single subkind declarations.

```
class Base;
```

Again, not terribly interesting, although this is a new possible use of the `class` keyword. Additionally, this brings up an important point related to kinds. As one could probably assume, kinds were very much static objects in Links until this project. This is not very helpful for what is being added because it is explicitly saying that subkinds are not *always* known when you are

parsing a file. I chose to look at this hurdle similar to a familiar concept: a type variable. So now kinds are represented via a sugar module, `SugarKind`. In this case, it is syntactic sugar because it has no real meaning, as it is yet to be confirmed that it exists. There are two options for a sugar kind:

```
type t =
  | KResolved of PrimaryKind.t option * Subkind.t option
  | KUnresolved of
    PrimaryKind.t option *
        (Linearity.t option * Restriction.t option)
```

Either it is "resolved" or unresolved. Unresolved has many options because subkind and kind annotations have two different situations in which they can appear. In `typename` calls they are technically kinds because both `typename Foo(a::Row) = ...` and `typename Bar(b::Base) = ...`. While in a signature datatype, the primary kind is specified with specific notation (`a` is `Type` and `{ a}`— is a `Row`), so it is only possible to specify subkinds in that case. This means the latter can resolve the primary kind at the parsing stage (unless of course kind inferencing becomes available), while the former cannot. A subkind's quantifiers are like a `typename`, so there is more flexibility and therefore more possible combinations of kinding information.

As is later demonstrated in the subkind environment, the primary kind is left optional in sugar kinds because there could be occurrences of subkinds that don't have an explicit primary kind. As is the case with `Base` and `Any`, both can be used as subkinds for any primary kind, while some subkinds like `Eff` should only be used with kind `Row`.

A quick note: A class is making use of type variables; however, in the implementation, a type variable is stored in the sugar phase as a `SugarQuantifer`, which is resolving both kind and freedom. Freedom can either be flexible or rigid. They are referred to as quantifiers because in the desugaring process, Links is determining the quantification of a given variable, which is closely linked to subkinds, as they act as the guiding hand in determining the scope, in terms of types, of a class's functions. Of course, now quantifiers make use of sugar kinds and therefore must resolve kinds before type variables can properly be resolved.

As demonstrated already, a class's type variables can be annotated with `::Type`, which will be marked as a resolved sugar kind or, if a subkind is used as in the subclass example, as unresolved with a restriction. The latter will then have to be ensured a kind or if kind inferencing is enabled, then inferred. By default, an unresolved kind without a primary kind will be assumed to be of `Type`. Finally, a type variable could be marked as `::Type(Unl, Any)`, which is a "full" kind definition because the primary kind, linearity and restriction have been specified. That kind can be marked as resolved during the parsing stage but will still have to confirm its restriction actually exists.

Returning to the sugar representation module, alongside the class block, there is also `ClassDecl` and `ClassFun`. The former tracks the name of the subkind class as well as its type variables or quantifiers. They are properly quantified in the desugaring type variable pass and then propagated to each class function. The class function is represented by `ClassFun of Class.single Class.t`, which is represented by the same module as the class block. However, it is now marked as single and thus contains a single operator, datatype, and reference to its class's name. These class function binding nodes are only created in the desugaring of subkind classes pass.

The last of the new sugar representations is the instance, which is denoted as:

```
Instance of Name.t * datatype' *
    (Name.t * (SugarQuantifier.t list * tyarg list *  phrase)) list
```

Quite simpler than the class declaration code, all that is required is the class name, the type the instance is instancing, and, of course, the functions that must implement the subkind class declarations. Operator bodies are marked as phrase, which allows the use of variables, lambda functions, constants, and other possible expressions. A function's type variables through multiple passes adopt the generalized signature of the related operation declaration and then instantiate that signature later with the instance type.

6.4 Desugaring Passes

Links, while comparably much simpler that Haskell, still boasts an impressive amount of features, all of which need to be translated into something meaningful on the backend. For that reason, there are a lot of desugaring passes that translate the syntactic sugar into an internal representation. This is not to be confused with IR or intermediate representation, which can be a data structure or a language of its own that represents source code for the compiler. The constraints, for example, are comparing and working alongside types as they are represented in Links, following the desugaring of type variables. Almost synonymous with desugaring is resolution, as in the case of type variables where proper quantification is discovered and used in the following stages.

There is not too much desugaring to cover, and the most important and interesting parts are in the desugaring of type variables, which now also performs the desugaring of kinds as well.

6.4.1 Modules

That being said, desugaring modules is where class functions are transformed into their readable representation. On the same token as OCaml, a module can refer to either the construct module or a file. Even with a single source file and no modules, this pass occurs, and in the case of the prelude, all the objects of that file are by default open imported to a Links program. Open import simply means that a source file's code is included without the necessity to prefix anything.

Here is where the call is placed to the new desugaring subkind class pass, which deconstructs a subkind class into two types of sugar types: a class declaration and class functions. Depending on the mode Links is in, this is either performed on the program level or the sentence level, which is how code is interpreted when using REPL mode. The class functions are top level bindings or global, meaning that they can be used anywhere; however, they still need to be processed at this point.

6.4.2 Type variables

The next pass that is relevant for the subkind classes is the desugaring of the type variables. This stage of desugaring is focused on resolving type variables and reporting errors during that process. Resolved type variables contain unionfind points, and if another unresolved type variable is found to be referring to this already resolved variable, then it will assign this same unionfind. Furthermore, unbounded type variables where no named variables should be bound implicitly will also be reported as errors at this stage. Additionally, sugar quantifiers and sugar kinds are resolved.

Information about type variables is stored in a map, which has both information in sugar kind and unionfind point form. The kinding information in the former is optional, while it is guaranteed to be resolved in its latter form. It's indicated that the sugar kind form takes precedence because there is the possibility of an absence of kinding information that is yet to be ensured or inferred in some cases. However, upon resolution, it must match the resolved formed present in the map. A new map has been created: `type subkind_classes_map = subkind_class_info StringMap.t`, which stores information related to subkind classes.

```
type subkind_class_info = {
  primary_kind: PrimaryKind.t option;
  subkind: Subkind.t;
  quantifiers: SugarQuantifier.t list;
}
```

This map provides a way of interacting with declarations and their related function signatures. All of this information is fulfilled by a class declaration.

Class declaration

```
| ClassDecl (name, qs) ->
  let o, name = o#name name in
  let make_parents rest =
    match rest with
    | Some r -> [ r ]
    | None -> ["Any"]
  in
```

First, simply the name of class is bound as a binding node, and then a simple function is made that figures what parents or ancestors the restriction of the class has.

```
let o, unresolved_qs, shared_kind =
    match qs with
    | _ ->
      let o, unresolved_qs = o#list (fun o q ->
        let (tyvar_name, kind, fd) =
            SugarQuantifier.get_unresolved_exn q in
        match kind with
          | SugarKind.KResolved (pk, sk) ->
            let class_subkind = (match pk, sk with
                | _, None ->
                    Restriction.add ~parents:["Any"] name;
                    (lin_unl, name)
                | _, Some (lin, rest) ->
                    let parents = make_parents (Some rest) in
                    Restriction.add ~parents name;
                    (lin, name)
                ) in

            let kind =
                SugarKind.mk_resolved pk (Some class_subkind) in
            o, SugarQuantifier.mk_unresolved tyvar_name kind fd
```

We'll ignore cases with no type variables, which can be assumed will always be those subkind declaration statements highlighted before. The parser will require a subkind class use at least one variable. What begins here is for each variable and quantifier is being constructed using the subkind information. This first match is for subkinds that have already been resolved because they are either primary kinds or full subkind statements. The only assumption made here is that when no linearity is indicated, the subkind is defaulted to nonlinear. In that same instance, the ancestors of that subkind will be set to the most general subkind parent, `Any`. On the other hand, when a subkind is identified for a type variable, then that restriction is made the parent and the linearity is reused. Either way, a new restriction is created of the name of the class, and that quantifier's kind is updated to use the new restriction. The sugar kind here was resolved before, and so it can be safely marked as resolved again. It is necessary to update the quantifiers subkind information as those are the quantifiers that will be used by the overloaded operators.

```
| SugarKind.KUnresolved (pk, (lin, res)) ->
    let parents = make_parents res in
    Restriction.add ~parents name;

    let kind = SugarKind.mk_unresolved pk (lin, Some name) in
    let q = SugarQuantifier.mk_unresolved tyvar_name kind fd in
    o, q
    ) qs
in
```

In the following case, the sugar kind has not been finalized. Utilizing that function from before the parents to be added will either be the provided restriction or, if there is none, then default to `Any`. Again, the quantifier is updated with the new restriction name; however, this kind information will have to be resolved at a later point.

```
let shared_kind =
    let sugar_kinds = List.map
        (fun q ->
            SugarQuantifier.get_unresolved_kind_exn q) unresolved_qs
        in
    let pks = List.map
        (fun k -> SugarKind.get_unresolved_pk_exn k) sugar_kinds in
    let sks = List.map
        (fun k -> SugarKind.get_unresolved_sk_exn k) sugar_kinds in
    let pk =
      if pks <> [] then
        match ListUtils.find_fstdiff pks with
        | Some _ ->
            None
        | None ->
            (List.hd pks)
      else
        None
    in
    let sk =
      if sks <> [] then
        match ListUtils.find_fstdiff sks with
        | Some _ ->
            failwith "Mismatch in Subkind class's linearity"
        | None ->
          (List.hd sks)
      else
        Some (lin_unl, name)
    in
    pk, sk in o, unresolved_qs, shared_kind
in
```

The second to last step of the desugaring of the class declaration is a check that attempts to find a shared kind. From all the class's quantifiers a list of both primary kinds and subkinds are created. If there is not symmetry between all the kinds then no primary kind is indicated, if there is then a shared primary kind is determined. Similarly for subkinds of the quantifiers, however, if there is some discrepancy then it should not be accepted. This bars cases where type variables of a class use different sorts of linearity, as restrictions will all have updated to match by this point. The final step is to add this new subkind class to the subkind class environment, which utilizes the same schema for environments that aliases use.

Each declaration encountered along the way is added immediately to the mutual environment of subkinds. Utilizing the `shared_kind` function an optional primary kind and required subkind are retrieved in order to be bound to the name of the class. The class's quantifiers and map are left empty for now because the former is yet to be resolved and the latter has no entries yet. Hence, the name mutual environment, this binding to a subkind environment is local to this pass and is only needed to verify the existence of subkinds used in type variables. The same thing was already implemented for aliases in the datatype pass because signatures could use aliases that are define in source code and not yet full translated internally. The subkind environment will be fully completed later.

```
let pk, sk, subkind_env =
    let (pk, sk) = shared_kind in
    pk, sk,
    SEnv.bind name
        (`Class ((pk, sk), [], StringMap.empty))
        subkind_env
in
let info = {
    primary_kind = pk;
    subkind = sk;
    quantifiers = unresolved_qs
} in
let o = o#set_subkind_class name info in
let o = o#set_subkind_env subkind_env in
(o, b)
```

A side note: To maintain the current code, a module that defines default subkinds has been added. This makes use of the simpler declaration subkind that lacks both quantifiers and the map of operation signatures.

```
let subkind_env : Types.subkind_environment =
  List.fold_left
    (fun env (name, t) ->
      SubkindEnv.bind name t env)
      SubkindEnv.empty
    [
      "Mono"      , `Decl (None, (lin_any , "Mono"));
      "Any"       , `Decl (None, (lin_any , "Any"));
      (* for linear effect vars *)
      "Lin"       , `Decl (None, (lin_unl , "Any"));
      "Base"      , `Decl (None, (lin_unl , "Base"));
      "Session"   , `Decl (None, (lin_any , "Session"));
      "Eff"       , `Decl (Some pk_row , (default_effect_lin , "Eff"));
    ]
```

These specifications were included based on documentation in the parsing file, but there was not a clear indication of what subkinds had what attributes. As can be gathered so far, some of this prerequisite work involved better defining kinds and allowing them to be dynamic objects intended for user extension.

Class functions

By this point, all of a class's functions have been desugared into single units for easier manipulation, and the first step is to fetch a function's name and the class's name and to then fetch a class's map from before. Of course, if there is no map found, then no class has been created, and a declaration is guaranteed to be processed before the first of its classes.

```
let class_name = (Class.name f) in
let fun_name = Binder.to_name (fst (Class.fun' f)) in
let class_info =
    try StringMap.find class_name subkind_qs_map
    with NotFound _ ->
        raise (internal_error
            ("Subkind class not found: " ^ class_name))
```

At this point, a function's type variables "boundness" is determined. A function's datatype (or signature) is properly quantified using the unresolved quantifiers that were entered previ-

ously. What the object method `quantified` does is quite complicated; it retains the original environment `o` before binding and then unbinding a list of quantifiers. Those quantifiers are then resolved and returned.

```
let o = o#reset_vars in
(* Process the class *)
let o, resolved_qs, fun' = o#quantified
    ~rigidify:false class_info.quantifiers (fun o' ->
      let b, dt = (Class.fun' f) in
      let o', b = o'#binder b in
      let o', dt = o'#datatype' dt in
      o', (b, dt))
```

This method is used for `forall` and `typename` statements as well, so in all cases, this operation is performed on type variables of a datatype. With the environment preserved, an unresolved quantifier is confirmed to be unique (all type variables are assigned a number in sequential order), and then two kinding checks are performed. The first one was in Links to begin with, and in cases where no primary kind is attributed to a type variable yet, the primary kind `Type` is set. However, using an optional config to infer kinds will instead forgo the addition of a primary kind.

The second check is a new one that I added to ensure a kindness is actually applied. This is of course needed, as subkinds can now be introduced by a user. This function covers all possible combinations of kinding information and makes use of the mutual subkind environment to resolve kinds where is possible. Unlike the previous ensuring of kinding, this function does not default to anything; rather, it is a frontend of sorts for the subkind environment. So in cases where no primary kind is indicated and a subkind has no specific kind requirements, then only a subkind is resolved. Furthermore, in those cases, it simply requests the linearity of the subkind. In the end, it also checks that no misuse of linearity or primary kind occurs in this unresolved kind; if so, it will fail with the appropriate error.

Following these checks, the rest of the binding function sets the freedom (rigid or flexible) of a variable before properly binding it to the context. Notice that before the class function's call

to quantified, a reset variable call was made; as one could guess, that was done to ensure that type variables from a previous inference environment do not leak into the current scope.

Unbinding intuitively does the same thing in reverse with the goal of fetching that unionfind point from before, which is the resolved representation of the quantifier. The point is then to bind type variables to an environment with varying levels of bounds, based on subkinds, and then to perform the opposite on the preserved original environment to check if there is no a resolved type variable in the current context with the same name, confirming that the binding of the quantifier was performed properly. And with that, the resolved quantified type variables can be returned alongside the new environment.

Practically what occurs is the unresolved sugar quantified type variables are applied to a function signature and thereby get resolved. For example, let's say a class `Num` has a single type variable that will in the declaration pattern be figured to be of kind `::(Unl, Num)`. Then a signature for an overloaded operation is read: `sig + : (a, a) -> a`. Currently, the variable `a` is too wide and needs to be constrained. Thus, the quantification operation is performed using both a primary kind and subkind and produces the output:

```
Class fun `+` has quantifiers:
        (380,
        (CommonTypes.PrimaryKind.Type,
        (CommonTypes.Linearity.Unl, "Num")))
```

What is happening according to Hindley-Milner is a processing of polytypes. Polytypes are types that contain variables bound by zero or more $\forall$ quantifiers. To reiterate the `id` function once again, its signature is represented in theory as: $\forall\alpha.\alpha \rightarrow \alpha$; where as a polytype any value of the same type to itself can be properly mapped to α. In fuller form, this type can be represented as $\forall\alpha_1...\forall\alpha.\tau$, where the $\forall$ is the quantifier that binds type variables in the monotype τ. A monotype is a type constant like `Int` and type variables are admitted as monotypes before eventual resolution. Variables α_i are referred to as *quantified* and all occurrences of the quantified type variable in τ is thusly bound. Unbound variables in τ are called *free*. If a type variable occurs in the context and is quantified in a polytype, then those variables will behave like type

constants. Additionally, type variables can appear unbounded while type checking and will be implicitly quantified to the most general quantification.

It is not the case that there will be a mix of bound and unbound type variables in any case. Type variables without a subkind will be treated as all-quantified variables implicitly. This can be seen with a signature that leaves its variables as `(a, a) -> a`. Therein is exactly what we want to avoid with the class's function signatures. So borrowing the quantifiers from the class declaration and using them in this "pre-processing" of type variables, a proper quantification is applied and each signature adopts the bounds of the class. This is not generalization yet, as that is performed much later. For the purpose of subkind classes, this just propagates the correct bounds to the rest of the type variables in function signatures.

The final thing do in the processing of a class function's signature is to update a class function with the resolved quantifiers. Notice that the map entry for this subkind is not updated with the resolved quantifiers, as they still need to be used in quantifying the rest of a class's functions. At the end of this pass, that map is thrown out, and all signatures have been properly bounded.

6.4.3 Datatypes

Desugaring of datatypes is considerably simpler. The class declaration will at this point create the new dynamic constraint and add it to the environment. Class functions will perform a general desugaring of their datatypes, which looks the same across other constructs as well. All that is happening here is that the syntactic representation of a datatype, such as a function signature, is translated into the internal representation. This time around, instances of subkind classes will be considered, and their datatypes, as well, go through the same desugaring process. Additionally, the instance will place a call to update the associated constraint with its datatype. For example, if it were, say, `instance Num : Int` then the constraint of the class is updated to allow operations to be performed on integers.

6.5 Transforming Sugar

With all relevant sugar constructs properly desugared, there are a few cases where further work is required before this can be properly compiled for use. By this point and going forward, all instances of `Class` should have been deconstructed by now, and all that is left is to transform declarations, function signatures, and type instances.

Class function signatures need to be added to the subkind environments map of functions; this makes it easier when constructing the dispatch table later on as well as error checking instances. All that is updated in terms of the map is this single function's name and signature is added to the map and kinding information and available quantifiers remains the same. Unlike before, this is the final state of the subkind environment and is utilized for instances of a class to fetch this information after it is filled out. In this same module, the class declaration case adds the appropriate quantified type variables.

```
let old_map = lookup_subkind_map context class_name in
let pk, sk = kind_of_subkind_map old_map in
let qs = qs_of_subkind_map old_map in
bind_subkind env
    (class_name, `Class ((Some pk, sk),
    qs, (StringMap.add (Binder.to_name binder) datatype old_map)))
    ) env ((Class.name f), (Class.quantifiers f))
in
( ClassFun (Class.modify ~funs:[(binder, (dt, Some datatype))] f)
, env
, Usage.empty )
```

For an instance of a class, the first thing is to check if the subkind is properly registered in the environment in order to fetch the quantified function signatures.

```
let _pk, _sk, _qs, funs =
    match lookup_subkind context class_name with
    | Some ( (pk, sk), qs, methods ) ->
        pk, sk, qs, methods
    | None   ->  failwith ("Class not found: " ^ class_name)
```

After a subkind is confirmed to exist, then another check is done in order to confirm that all signatures are being matched with a respective implementation in this current instance. This is a simple check between the operators that are required and the names of the ones in scope.

```
let req_ops =
  let list = Utility.StringMap.to_alist funs in
  List.sort String.compare (List.map fst list)
in
let new_ops =
  List.sort String.compare (List.map fst instances)
in
if List.equal (String.equal) req_ops new_ops then
  ()
else
  List.iter ( fun op ->
      if not (List.mem op new_ops) then
        raise (Errors.unbound_subkind_instance pos class_name op)
    ) req_ops
```

This where the deduction of the subkind classes occurs in the code. Using the Hindley-Milner's set of rules, it can be easy to translate the meaning of certain objects. The explanation of such will use the proper terms but will not lean too heavily on the theory, as these function calls are quite abstracted. In broad strokes, first the datatype is derived for this instance by the argument held in the instance construct, going for using `Int` as the example and the function signature it is fulfilling is `+`. This means we want to take the retrieved signature `(a::Num, a) -> a` and rewrite the type as `(Int, Int) -> Int` for this instance as well as type check the body that is provided. The body of an instance can either be a variable pointing to a function or an inline lambda as shown previously. Additionally, a body could be any sort of expression, so constants and conditional statements would work as well.

The formal rules for deduction of a Hindley-Milner system, which will be referenced for the remainder of this section, are included in the appendix.

```
let datatype =
    match dt with
    | (_dt, Some datatype) -> datatype
    | _ -> assert false
in
let instances =
    List.fold_right
    (fun (op, (_qs, _tyargs, body)) bindings ->
    let map = lookup_subkind_map context class_name in
    let dt' = StringMap.find op map in
    let dt' = Instantiate.freshen_quantifiers dt' in
    let (tyargs, dt') = Instantiate.typ dt' in

    let body = tc body in
    let bt =
      unify pos ~handle:Gripers.bind_val_annotation
        (no_pos (typ body), no_pos dt');
      dt'
```

The first step is to get access to the type of the function; in this case, the scope is limited to this single implementation, and, therefore, the only two types within that context is the signature fetched via `lookup_subkind_map` and the body that is to be unified. Working with the example provided earlier, let's say we are attempting to add the instance of type `Int` for the addition operator.

$$\frac{\Gamma \vdash_D (+) : \forall \alpha : Num.\alpha \to \alpha \to \alpha}{\Gamma \vdash_D (+) : \forall \alpha <: Num.\alpha \to \alpha \to \alpha}[Var]$$

It can be concluded that `dt'` has the function type, quantified by the subkind `Num` within the context. Then to actually make use of this polytype statement, it must instantiated:

$$\frac{\Gamma \vdash_D (+) : \forall \alpha <: Num.\alpha \to \alpha \to \alpha \qquad \forall \alpha <: Num.\alpha \to \alpha \to \alpha \sqsubseteq Int \to Int \to Int}{\Gamma \vdash_D (+) : Int \to Int \to Int}[Inst]$$

This follows a simple rule that if the type of our polymorphic function, within the context, contains the integer function type of the same kind, then it can be instantiated. In other words, replace type variables with the applicable datatype. And a similar logic would apply for multiple types such as an instance of a pair of type variables `(a, b)` in that each kind of variable would have to be checked to see if it contains the datatype that is attempting to be applied. To freshen

the quantifiers means simply to create a fresh variable in order to prevent variable clashes or later reuse by other constructs.

Then the body is type checked and a unification is attempted between the body and this instantiated datatype. This serves as a check to verify that a supplied body is actually of the type that you are instancing. In a sense, it is verifying the contract that you have stated with `instance Num : Int`. Assuming you have provided something akin to `addInt`, it can successfully unify and continue along. Hidden behind the abstraction, there is usage of what it is called let polymorphism—a core feature of Hindley-Milner. This is used implicitly here to allow polymorphic types to propagate properly, avoiding reliance on a basic lambda abstraction rule, which does not support polymorphic types. This is something that system F has solved as an extension to Hindley-Milner; however, this is outside the scope of the purpose of this paper.

```
let _usage = usages body in
let body = erase body in
let sugar_tyvars = List.map SugarQuantifier.mk_resolved tyvars in
(op, (sugar_tyvars, tyargs, body)) :: bindings)
instances []
in
(Instance (class_name, dt, instances)
    , empty_context, Usage.empty)
```

The final step discards the usage annotations and marks the type variables as fully resolved before adding this new implementation to a list of bindings for the instance. This previous operation is performed on all implementations of an instance before proceeding.

6.5.1 New errors

Noted before, there are new errors within the Errors module related to classes. The unbound subkind error can be fired off during the desugaring of type variables and kinds in the case of encountering a subkind that does not exist in the environment. Additionally, there is an error for instances when they fail to properly define implementations for all signatures of a class. This is to ensure there will be no runtime complications of unresolved functions for a type that should have one.

6.6 Translation to IR and Runtime Resolution

6.6.1 Sugar to IR

If you are following this in a linear order, then you should know that by this point, constructs have both been desugared and transformed. Now all that remains before the proper resolution of overloaded functions is to translate to the IR or intermediate representation. Because Links is interpreted, the actual evaluation of the IR will happen at runtime, so this is the final step before then. Fatal errors are still caught at this point, but type errors should not happen at this point.

```
| CFun      of binder
| CInst     of binder * (language * tyvar list * tail_computation)
```

Both the function signatures and instances are represented as the following in the IR module. A `binder` should be confused with a `Binder` from the sugaring phase. The `binder` here is a reference to a variable in the environment. A noted area of improvement in Links is that top-level variables are treated as binds along with functions. Of course, all functions are top level. This construct tracks a variable number, the actual operation name and a datatype. Furthermore, in `CInst`, only a single unit is tracked. Its binder tracking the type of the argument, language, the operation and `tail_compuatation` is the body from before but needs to be evaluated before the IR.

```
let binder = fst (Class.fun' fun') in
assert (Binder.has_type binder);
let class_name = (Class.name fun') in
let qs =
    List.map (fun q ->
    SugarQuantifier.get_resolved_exn q) (Class.quantifiers fun') in
let _type_args = List.map (fun q ->
    Types.type_arg_of_quantifier q) qs in
let f = Binder.to_name binder in
let ft = Binder.to_type binder in
let _f_var = snd (Var.fresh_var (Var.make_info ft f scope)) in
I.letcfun
(Var.make_info ft f scope, qs)
(fun v -> eval_bindings scope (extend [f] [(v, ft)] env) bs e)
```

Perform a simple assertion that the ad hoc operator has a type properly set for its binder, and then bind that operator to the environment. This is useful because it makes it clear to the program and the use just what exactly the type of the operator is. If one were to attempt to get the type of an operator by encapsulating it in parenthesis, it returns the correct type:

```
links> (*);
* : (a::(Unl,Num), a::(Unl,Num)) -> a::(Unl,Num)
```

However, this may be bound to the environment, but actually an implementation is needed upon application. Without any, then a runtime error would be thrown because when a constraint is created, it is created as empty and unable to fulfill any type. Type safety is maintained in that regard. So if no instances were actually added, then the previous would still work but upon application to anything a type error is fired.

```
let (_, datatype) = dt in
let rec process_methods remaining_methods scope env bs e =
match remaining_methods with
| [] -> eval_bindings scope env bs e  (* No more methods, continue *)
| (op, (qs, tyargs, body))::rest ->
    let x = ev body in
    let xt = I.sem_type x in
    let affix_op = op ^ Types.string_of_datatype datatype in
    let x_info = Var.make_info xt affix_op scope in
    let qs = List.map SugarQuantifier.get_resolved_exn qs in
    I.letinst (x_info, ec body, op,
       qs, fun v -> process_methods rest scope
              (extend [affix_op] [(v, xt)] env) bs e)
in
process_methods instances scope env bs e
```

For class instances, a recursive function has been created to process each implementation. Each body has an evaluation check before being bound to the environment as the operation name affixed with the datatype of instance. This was a convenient way of passing information along to the evaluation, but it is not a perfect solution. It does not overwrite the ad hoc operator because each name is distinct; however, it should not be necessary to populate the environment with additional operators. A different way of going about this would be to define the table in this section and, instead of binding to the environment, partition the operators and their bodies directly into that table for later use. Regardless, the final step before moving on is to take the

class function bindings and move them into the global environment. The same is done with all other function bindings as well.

```
| CFun bndr
    when Var.Scope.is_global (Var.scope_of_binder bndr) ->
    let f = Var.var_of_binder bndr in
    let f_name = Var.name_of_binder bndr in
    partition (b::locals @ globals, [],
     Env.String.bind f_name f nenv) bs
```

6.6.2 Resolution

It is at this point that proper functions are substituted for overloaded operators. For instance, adding two integers replaces `+` with `addInt` from the Lib module. This mirrors the approach taken in type classes, where overloaded functions are implemented via dictionary passing. This substitution is performed at runtime, following the reasoning given in Wadler's paper: In some cases, particularly when a function signature involves a subkind, the concrete implementation can only be determined at runtime, once the actual value is known [30]. Importantly, this substitution occurs only after full static type checking, ensuring that no runtime type errors can occur. During type checking, the class, its instances, and their application have all been verified. A representative example is the function `square`, which accepts a numeric instance and returns its square.

Whereas type classes rely on a compile-time dictionary that guarantees an implementation of the required type, subkind classes achieve this through constraint satisfaction. Both mechanisms rely on a form of runtime dispatch: In type classes, this is realized by translating a constraint like `Num a =>` into an additional dictionary argument (e.g., `numDInt 3`). For subkind classes, this means substituting the overloaded operator with its statically verified implementation at the point of use. Herein lies a weakness with the current implementation of subkind classes, as they are always dispatched at runtime. In cases where types are explicit during type checking, it would be appropriate to perform a replacement during compilation. However, this implementation does still work, and this is only a potential avenue for further improvement.

The dispatch table is represented using a hash:

```
module SubkindTable = struct
  let implementations = Hashtbl.create 50
  let add_impl name arg_type impl_var =
    Hashtbl.add implementations (name, arg_type) impl_var
  let find_impl name arg_type =
    try
      Hashtbl.find implementations (name, arg_type)
    with NotFound exn ->
      raise (NotFound exn)
end
```

Given an operator and argument type, return the proper function, which has already determined at this point to be the proper one.

```
| `SubkindClassFunction (name), args ->
let argtypes = List.map Value.typ args in
let impl_var =
try SubkindTable.find_impl name argtypes
with NotFound _ ->
    eval_error "Uncaught subkind class function %s." name
in
value env (Variable impl_var) >>= fun impl ->
apply cont env (impl, args)
```

When a subkind class function is applied, then this block gets matched, and a single lookup is performed. A not found is handled, but that is just reiterating a standard in this file; there are multiple checks before arriving here. With the correct implementation variable, a function is then applied, and the resulting value has been accounted for in its signature, and the whole process is now completed.

A few final points: To get to this runtime call of a subkind function, a `CFun` is registered just as primitive functions and function pointers are.

```
| CFun b ->
let var = Var.var_of_binder b in
let name = Var.name_of_binder b in
let scope = Var.scope_of_binder b in
computation
(Value.Env.bind var (`SubkindClassFunction name, scope) env)
    cont (bs, tailcomp)
```

With a class instance, rather than fully binding that operator to the environment, it is instead added to the table. Type constructs like lists, polymorphic variants, records, tuples, or rows

are first made concrete (remove any aliasing) and then added to the table. Their scope is then limited only to the places where they are applied.

```
| CInst (b, (op, _, tc)) ->
let var = Var.var_of_binder b in
let typ = Var.type_of_binder b in
let argtypes = TypeUtils.arg_types typ in
List.iter (fun argtype ->
SubkindTable.add_impl op (Types.concrete_type' argtype) var
) argtypes;
let locals = Value.Env.localise env var in
let cont' =
K.(let frame = Frame.make (Var.scope_of_binder b)
    var locals (bs, tailcomp) in frame &> cont)
in
tail_computation env cont' tc
```

7
Conclusion

Subkind classes are not without flaws, nor are they impervious to criticism. The goal of this project was to attempt to translate theory into practice and—at the grave risk of sounding too jovial—the journey was much more interesting than the conclusion. Furthermore, there is in no way an intent to proclaim that these solutions are superior to either previous closed or open ad hoc implementations. Those two ideas were used to illustrate the general notion of what can normally be done and then take what is relevant there and make it work in Links. The idea to rely on subkinding is not original to this paper but rather was utilized because it fits the *context* of the language. That is, where Haskell has more of a general view of types, Links has a stronger focus on kinds within its type system, utilized to differentiate between what data is coming and going. Of course, Haskell has kinds, but for Links, they have a vital importance to the whole project.

Making this a general system, that taps into this current interface with kinds, allows for greater control within Links proper, avoiding having to mess around too much with OCaml code. That being said, the first and simpler ad hoc solution is not necessarily worse. In actuality, its simplicity makes it a much easier choice for a pull request. And at the end of the day, no matter how interesting and useful it may appear to me, the reality of working on a project like this is that the ball is not in my court. Furthermore, both implementations were kept not only as

stepping stones for later parts but also because it simply boils down to ideological differences. OCaml has kept certain operators like `+` as single type operations, while others like `==` use a Standard ML approach. Of course, Haskell has fully adopted type classes and done a lot of great work since the original paper.

There are some drawbacks with subkind classes as they currently stand, besides opinionated ones. The most glaring is the utilization of a runtime dictionary, as well as no compile time resolution of applications of non-type variables to an ad hoc operator. The former could be solved relatively simply but was not implemented purely because it didn't have to be for the system to actually work. Bounded quantifiers ensure the type safety of the declarations and usage; therefore, compile time resolution is a route of possible optimization, not a way of ensuring better type safety necessarily. In terms of the former, some of these costs could be avoided using partial evaluation [9]. However, that is another project in and of itself.

While some issues remain and questions of possible tradeoffs are left unanswered, what was set out to be achieved was sufficiently implemented and illustrated here.

Appendix A
Additional Examples and Rules

A.1 Another Example of Subkind Inheritance

Again, making use of our classes `Ord` and `Eq`, we want to create a function `palindrome` that checks if a given list is the same reversed.

```
sig palindrome : ([%a::Ord]) ~> Bool
fun palindrome(l) {
    var rev = reverse(l);

    sig aux : ([%a], [%a]) ~> Bool
    fun aux(l, rev) {
            switch((l, rev)) {
                    case ([], []) -> true
                    case (x::xs, y::ys) ->
                            x `eq` y && aux(xs, ys)
            }
    }
    aux(l, rev)
}
```

Demonstrating that subkinds are able to make use of their parents' function definitions as long as an instance of that is made for that child's constraint.

A.2 Example of Alternate Type Variable Usages

Furthermore, more combinations of type variables can be used, such as the following. Making use of a pair of type variables that can convert between datatypes.

```
infixl 6 `coerce`;

class Coerce : (a, b) {
    sig `coerce` : (a) -> b;
}

instance Coerce : (Int, String) {
    `coerce` : intToString;
}

instance Coerce : (Int, Float) {
    `coerce` : intToFloat;
}

instance Coerce : (Bool, Int) {
    `coerce` :  fun(b) {if (b) 1 else 0};
}

instance Coerce : ([Int], Int) {
    `coerce` : max_list;
}
```

A.3 Implementation Code

Not all code has been included as indicated; however, some functions that are referenced during the discussion of their use and implementation have been provided.

A.3.1 Desugaring of Type Variables

```
method resolve_kind ?pos ~subkind_env unresolved_kind =
let open SugarKind in
let pos =
    OptionUtils.from_option SourceCode.Position.dummy pos in

let get_sk_info (name : string) :
    (PrimaryKind.t option * Linearity.t) =
  try
    match (SEnv.find name subkind_env) with
    | `Decl (pk, (lin, _)) -> pk, lin
    | `Class ((pk, (lin, _)), _, _) -> pk, lin
  with NotFound _ ->
    raise (Errors.unbound_subkind pos name) in

let pk_sk = match unresolved_kind with
| KResolved (pk, sk) -> (pk, sk)
```

```
| KUnresolved (None, (None, Some res)) ->
  let pk, lin = get_sk_info res in
  (pk, Some (lin, res))
| KUnresolved (pk_opt, (None, None)) ->
  (pk_opt, None)
| KUnresolved (Some pk, (Some lin, None)) ->
  (Some pk, Some (lin, res_any))
| KUnresolved (pk_opt, (lin_opt, Some res)) ->
  let class_pk, lin = get_sk_info res in
  let lin =
    match lin, lin_opt with
    | l, None -> l
    | l, Some l2 ->
      if l <> l2 then
        failwith
        ("Class requires linearity:" ^
            (Linearity.to_string l))
      else
        l
in
let pk = match class_pk, pk_opt with
    | None, None -> None
    | None, Some pk -> Some pk
    | Some pk, None -> Some pk
    | Some pk1, Some pk2 ->
      if pk1 <> pk2 then
        failwith ("Class requires primary kind:" ^
            (PrimaryKind.to_string pk1))
      else
        Some pk1
    in
(pk, Some (lin, res))
```

A.4 Formal Rules of Hindley-Milner

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash_D x : \sigma}[Var]$$

$$\frac{\Gamma, x : \tau \vdash_D e : \tau'}{\Gamma \vdash_D \lambda x.e : \tau \to \tau'} \quad [Abs]$$

$$\frac{\Gamma \vdash_D e : \sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash_D e : \sigma} \quad [Inst]$$

$$\frac{\Gamma \vdash_D e : \sigma \quad \alpha \notin free(\Gamma)}{\Gamma \vdash_D e : \forall \alpha.\sigma} \quad [Gen]$$

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Chapter 6 type checking*, Compilers: Principles, techniques, and tools, 1988, pp. 343 –386.

[2] Luca Cardelli and Peter Wegner, *On understanding types, data abstraction, and polymorphism*, Computing Surveys **17** (1985), 471–522.

[3] James Cheney, *"flat" and "nested" kinds for query results*, GitHub, 2022.

[4] James Cheney, Frank Emrich, Simon Fowler, Vashti Galpin, Daniel Hillerström, Sam Lindley, Wilmer Ricciotti, Wenhao Tang, and Philip Wadler, *The links programming language documentation : Kinds and subkinding*.

[5] ———, *The links programming language*, 2023.

[6] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop, *Links: Web programming without tiers*, Springer EBooks (2006).

[7] Evan Czaplicki, *Elm's types*, 2021.

[8] Sam Lindley Daniel Hillerstrom and KC Sivaramakrishnan, *Compiling links effect handlers to the ocaml backend*, ICFP (2016).

[9] Chris Fallin and Maxwell Bernstein, *Partial evaluation, whole-program compilation*, Proc. ACM Program. Lang. 9 (2025).

[10] United States Government, *Rationale for the design of the ada® programming language*, Ada Documentation (1983).

[11] Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh, *A classification of object-relational impedance mismatch*, 2009 first international confernce on advances in databases, knowledge, and data applications, 2009.

[12] Gabriel Radanne James Cheney Sam Lindley and Philip Wadler, *Effective quotation: Relating approaches to language-integrated query*, Edinburgh Research Explorer (2014).

[13] Simon Peyton Jones, 2017. In this talk Simon discusses Haskell's birth and evolution, including some of the research and engineering challenges he faced in design and implementation. Focusing particularly on the ideas that have turned out, in retrospect, to be most important and influential, as well as sketching some current developments and making some wild guesses about the future.

[14] Stefan Kaes, *Parametric overloading in polymorphic programming languages*, Proceedings of the 2nd european symposium on programming, 1988, pp. 131–144.

[15] Oleg Kiselyov, *Implementing, and understanding type classes*, Okmjj (2021).

[16] Anil Madhavapeddy and Yaron Minsky, *Real world ocaml: Functional programming for the masses*, 2nd ed., University Printing House, 2022.

[17] John Meacham, *Jhc*. A less popular alternative to GHC, JHC uses an intenstional type analysis system that derives from Kaes along with Harper and Morrisett's pioneering type-passing framework.

[18] John Stuart Mill, *A system of logic, ratiocinative and inductive*, University Press of the Pacific, 1843.

[19] Robin Milner, *A theory of type polymorphism in programming*, Journal of Computer and System Science (1978).

[20] ———, *Changes to the standard ml core language*, Laboratory for Foundations of Computer Science – University of Edinburgh (1987).

[21] Tim Newsham, *The curry-howard correspondence in haskell*.

[22] Lawrence C. Paulson, *Ml for the working programmer*, 2nd ed., Cambridge University, 1996.

[23] Benjamin C. Pierce, *Types and programming languages*, The MIT Press, 2002.

[24] Wilmer Ricciotti and James Cheney, *Query lifting: Language-integrated query for heterogeneous nested collections*, CoRR **abs/2101.04102** (2021), available at `2101.04102`.

[25] Dave Barton et al. Simon Peyton Jones Lennart Augustsson, *Haskell 98 language and libraries: The revised report*, Haskell Foundation, 2002.

[26] Christopher Strachey, *Fundamental concepts in programming languages*, Reader in Computation at Oxford University (1967), 11–49.

[27] Simon Thompson, *Type theory and functional programming*, Computing Laboratory, University of Kent, 1999.

[28] Jeffrey D. Ullman, *5.3 polymorphic functions*, Elements of ml programming, 1998.

[29] vcgalpin, *Roundtripping on negative floats #1070*, GitHub, 2021.

[30] Philip Wadler and Stephen Blott, *How to make ad-hoc polymorphism less ad hoc*, POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (1989).

[31] Henry Wandover, *Proper ad hoc solution for links*. The proper ad hoc solution for Links. https://github.com/yung-turabian/links/tree/subkind-overload.

[32] ———, *Subkind classes for link*. The subkind classes branch for Links, rebased so changes can be viewed in a single commit. That non-rebased version can be easily found from there. https://github.com/yung-turabian/links/tree/subkindclasses-rebased.

[33] ———, 2025. A small recording to demonstrate the result of adding two arrays in JavaScript.