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

Henry Wandover **Adviser: Professor Robert McGrail**

Computational Sciences Program, Bard College, 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.

Links

The Links programming language is the brainchild of the University of Edinburgh, a functional programming language designed to make web programming easier [1]. Referred to as the *impedance mismatch problem*, Links' call to fame is a single implementation language for all "tiers" of a web application. This means that interpreted code is translated into some of JavaScript for the frontend, bytecode for the server, and SQL for the database.

Ideally, the Links mindset serves as a sane method of programming for the web. Of course, a primary concern is data transfer/travel across the tiers; therefore, Links boasts a strong, static type system—as any good functional language should have. What distinguishes Links from a language like Haskell or OCaml, aside from the web-focused design of the language, is the utilization of *kinds* to distinguish what data is able to be read on the server versus the database.

Type

A type describes an expression; for example, $3 : Int$ or $'c' : Char$. In this case, there is a clear instantiated type, a type is so called instantiated if it begins as a type variable α and the type inference system is able to figure the type based on context.

Assume that there is a function id with the type signature $id : \alpha.\alpha \to \alpha$. This function could be said to be polymorphic—i.e. it accepts multiple types—as well as generalized. This is assuming that the function `id` has been added to the context. To apply, say, a string `"Bard"`, the function is fetched from the context by the rule `[Var]` and instantiated with the applied type, resulting, in this instance, in the function $\Gamma_D id : String \to String$, which can then properly allow the application (`[App]`) of `"Bard"`.

Kind

What is a kind then? A kind is simply a higher-order type describing parts of the universe of types. The most general kind is $*$ or `Type`, which is the kind of any datatype that does not have any type parameters. Take, for example, a list with the type signature of $[Int]$; however, in a function signature, it could make use of type variables; therefore: $[\alpha]$. In this latter state, it would have the kind $*\to*$, which should be read as it requires some datatype of kind $*$ to be able to construct a datatype. That example would be considered a unary type constructor. If a kind is considered a quantifier, in the `id` example, it could have been more explicitly written as: $id : \forall\alpha.\alpha \to \alpha$, which indicates that α is universally quantified. More specifically, in Links, that would mean any type of kind $*$ could be used. That is important for parametric polymorphism, but for ad hoc, there is a necessity to constrain the quantifier.

Subkind

Therefore, Links makes use of subkinds, which can optionally have a specific "parent" kind. There are three kinds in Links: `Type`, `Row`, and `Presence`. One often used subkind is `Base`, which can be used as a constraint on any kind. An example given is when querying the database by calling `fun (e) {query {e}}`, the subkind received is `Base` and is furthermore inferred by the type system. This is, of course, an extension of Hindley-Milner type system, and the ad hoc work of this project further extends from the vanilla system.

Rather than using subtyping, as the ML-style ad hoc polymorphism does [2], or creating compile time dictionaries that utilize abstract datatypes [3], the decision was made, for both implementations, to utilize subkinding as the generalized system of constraint to bound the ad hoc in a type safe way. In both cases, any sort of misuse of an overloaded operator is caught at the type checking phase as not having a reference to the proper operation for a given type.

Polymorphism

A canonical example of parametric polymorphism is the function `length`, which has a signature of $length : \forall\alpha.[\alpha] \to Int$. Notice that this takes a list, a type constructor, and while a type is instantiated for the type variable, the actual type does not matter for the operation. All that matters is the number of elements, and, therefore, the implementation always looks the same—popping the head off and recursively counting until the end.

Consider the operation: $(+) : [\alpha :: Num].\alpha \to \alpha \to \alpha$, which utilizes the subkind `Num`, *not* the universal quantifier as in the previous case. Therefore, the contract stated here is that `+` has underlying implementations for any type of the subkind `Num`. This is referred to in the paper as a case of *bounded* polymorphism due to the checking of a subkind's constraints before approving an application.

Usage

The first implementation is an ML-style closed ad hoc system that overloaded arithmetic operations to comply with values of type `Int` and `Float`. Of one of the included examples, `Num` is demonstrated to be easily created within Links rather than OCaml (its implementation language). That demonstrates the user-extensible system of ad hoc supported via subkind classes, where declarations are made with `class` and then widen the constraint with new `instances` of a class.

```
class Num : (a::Type(Unl, Any)) {
        sig + : (a, a) -> a;
        sig * : (a, a) -> a;
}
instance Num : Int {
        + : addInt;
        * : mulInt;
}
instance Num : Float {
        + : addFloat;
        * : mulFloat;
}
```

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

sig checkTempEquality : (Temp, Temp) -> Bool
fun checkTempEquality(x, y) {
    switch ((x,y)) {
            case (Cel(m), Cel(n)) -> m == n
            case (Far(m), Far(n)) -> m == n
            case (Cel(c), Far(f)) ->
                (1.8 * c + 32.0) == f
            case (Far(f), Cel(c)) ->
                (1.8 * c + 32.0) == f
    }
}
instance Eq : Temperature {
    == : checkTempEquality;
}

Cel(10.0) == Far(50.0)
```

References

[1] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop, *Links: Web Programming Without Tiers*, Springer EBooks (2006).

[2] Robin Milner, *Changes to the Standard ML Core Language*, Laboratory for Foundations of Computer Science – University of Edinburgh (1987).

[3] 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).