

GeckoGraph: A Visual Language for Polymorphic Types

Shuai Fu, Tim Dwyer, Peter J. Stuckey, John Grundy

Monash University, Australia

Abstract

Polymorphic type is an important feature in most strongly typed programming languages. They allow functions to be written in a way that can be used with different data types, while still enforcing the relationship and constraints between the values. However, programmers often find polymorphic types difficult to use and understand, and tend to reason in terms of concrete types. We propose GeckoGraph, a graphical notation for types. GeckoGraph aims to complement traditional text-based type notation by making reading, understanding, and comparing types easier. We conducted a large-scale user study comparing GeckoGraph to text-based type notation. To our knowledge, this is the largest controlled user study on functional programming ever conducted. The results show no overall significant differences in task completion time or success rate. However, we observed a modest improvement in success rate among beginner participants. For the most challenging task, GeckoGraph improved success rates by 14.5% for beginners and 11.2% for less experienced users. These findings suggest that GeckoGraph has potential as a teaching and learning aid for polymorphic type systems.

Keywords:

Polymorphic Types, Functional Programming, Visual Languages, Visualization

1. Introduction

In programming languages, a polymorphic type [1] can represent values of different types while providing an interface to describe common behaviors for those values. Polymorphic types are one of the oldest topics in programming language theory [1]. They are central to the succinctness of statically typed languages while also enabling a high degree of type-safe abstraction and reusability. Polymorphic types are available in many programming languages, from functional languages such as Haskell and ML to imperative and multi-paradigm languages such as Rust[2] and Go[3].

While promoting robustness and code reusability, polymorphic types present challenges in learning and comprehension, particularly for novice users [4, 5]; as it is often argued that the expressiveness of type systems often comes at the cost of their usability [6]. Research indicates that humans tend to focus on concrete types and rely on polymorphic type checking only as a last resort. Moreover, polymorphic types often complicate the usability of type errors. During type checking, compilers frequently generate new polymorphic type variables temporarily. These variables are typically discarded after type checking unless a type error occurs, in which case they are referenced in error messages. Consequently, programmers often face the task of resolving

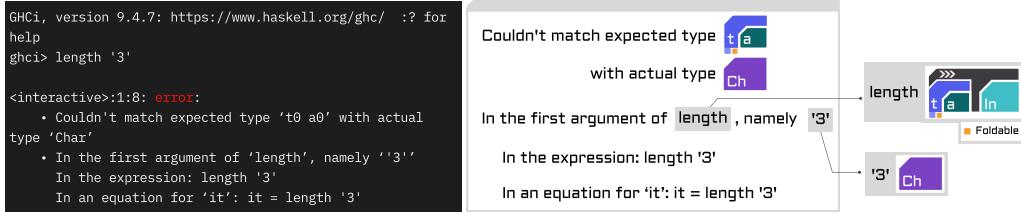


Figure 1: An example type error where a programmer mistakenly provided a `Char` instead of a `String` literal. **Left:** Original type error. **Right:** Type error with GeckoGraph

type errors involving type variables they did not explicitly author (Fig. 1). Unfortunately, not many studies focus on the *usability* of polymorphic types.

We aim to investigate the challenges of using polymorphic types and explore how to improve their usability with visualization and modern Human-Computer Interaction (HCI) techniques. In this paper, we propose GeckoGraph, a graphical notation for types. GeckoGraph aims to complement traditional text-based type notation and make reading, understanding, and comparing types easier. GeckoGraph is prototyped and iteratively verified, resulting in a design with visual clarity applicable across many programming contexts. To evaluate its effectiveness, we conducted a large-scale user study comparing GeckoGraph with standard text-based type notation. To our knowledge, this represents the largest controlled user study on functional programming to date. The results show no overall significant differences in task completion time or success rate. However, we observed a modest improvement in success rate among beginner participants. For the most challenging task, GeckoGraph improved success rates by 14.5% for beginners and 11.2% for less experienced users. These findings suggest that GeckoGraph has potential as a teaching and learning aid for polymorphic type systems.

In Section 2, we review key type system features in functional programming languages, many of which have recently been adopted by object-oriented and multi-paradigm languages. We highlight how these features enable flexible abstraction and expressive type modeling, while sometimes hindering usability for beginners. Section 3 presents the design of GeckoGraph, illustrating how it supports the type level features outlined in Section 2. Section 4 describes our user study, ZeroToHero, a series of 10 programming challenges, along with the study design, recruitment, and hypotheses. Section 5 reports quantitative and qualitative findings, including completion time, success rates, and participant feedback. In Section 6, we interpret these results, discuss GeckoGraph’s benefits and usability challenges, and explore potential applications. Section 7 reviews related visualization research, and Section 8 summarizes our contributions and outlines future directions.

2. Type Systems in Functional Programming

Functional programming languages have a rich history of embracing strong type systems and pioneering advanced type system features [7, 8, 9]. In functional programming languages, type systems are often more expressive and provide stronger safety guarantees. They enable programmers to model program behavior rigorously and enforce correctness at compile time. Common characteristics of type systems in functional programming include structural typing, algebraic data types, parametric polymorphism, and type inference.

```
main.ts
```

```
const x = { a: { b: { c: { d: 3 } } }, e: { f: { g: 4 } } }
function f(y: { a: { b: { c: { d: string } }, e: { f: { g: number } } } }) {
    return y.a.b.c.d
}
f(x)
```

```
~:$ tsc main.ts
```

```
ERROR Argument of type '{ a: { b: { c: { d: number; }; }; e: { f: { g: number; }; }; }' is not assignable to parameter of type '{ a: { b: { c: { d: string; }; e: { f: { g: number; }; }; }; }'. Types of property 'a' are incompatible. Property 'e' is missing in type '{ b: { c: { d: number; }; }; }' but required in type '{ b: { c: { d: string; }; e: { f: { g: number; }; }; }; }' typescript(2322).
```

Figure 2: Example of a TypeScript structural type error. (Top) The function `f` expects a parameter with a string field `d`, but the argument `x` provides a numeric `d`. (Bottom) The error message is dreadful to read.

Structural typing is often the primary approach for determining type equivalence and compatibility in functional programming languages (e.g., Elm and OCaml) and multi-paradigm languages (e.g., TypeScript). In contrast, nominal type systems are often predominant in Object-Oriented languages such as Java and C# [10]. In nominal systems, two types are considered distinct even if they have identical structure but different names. In contrast, structural type systems consider types equivalent as long as their structures match, regardless of their names. This approach promotes flexibility, as types can be defined inline without introducing new names, and values are automatically valid for a type when they conform to the structure. However, this lack of nominality can also make complex types harder to read and reason about. For instance, when encountering type errors, programmers can face two mismatched types with a complex structure, and understanding where exactly the root error is becomes a tedious task of string comparison (Fig. 2).

Algebraic data types (ADTs) [11] allow developers to construct complex types from simpler ones using sum and product operations. A **sum type**, often called a tagged union, $A + B$, represents a type that can be a value of A or a value of B . Conversely, a **product type** $A \times B$ represents a tuple type composed of two values of both A and B . This formalization allows precise and composable type construction for common data structures such as lists, tuples, and result types. For example, in Haskell, types like `data Either a b = Left a | Right b` and `data Maybe a = Nothing | Just a` provide basic building blocks for error handling, and the type `data List a = [] | a : List a` provides a succinct recursive definition for lists. This illustrates the expressive power of ADTs. However, in practice, type definitions can grow to be complex structures, for instance `Maybe (Either [Maybe a] (Either a String))` as programs grow, causing difficulties to interpret and maintain.

Parametric polymorphism (often called generics) allows functions and data structures to operate uniformly across different types. For instance, in Haskell, the function `head :: [a] -> a`, taking a list as input and returning the first element, is a polymorphic function that works

on lists of any type; integers, strings, and functions are all valid candidates for the generic variable `a`. It provides a strong and principled form of abstraction, enabling code reuse without sacrificing type safety.

However, parametric polymorphism also increases cognitive load: programmers must mentally track type variables and reason about how concrete types are instantiated during function application. This task becomes more challenging when functions involve multiple type variables. An example would be the `foldM` function: `foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a`. The challenge of reading and understanding polymorphic types is more serious for novice users [4, 5].

Type inference, or implicit typing, reduces the need for explicit type annotations by allowing the compiler to deduce types automatically. Most modern languages employ some level of inference. For instance, Java’s `var` keyword (introduced in version 10) infers variable types at compile time based on their assigned values [12], saving programmers from writing redundant type declarations, such as `Car car = new Car()`.

Functional languages such as Haskell and ML go further, using global type inference [13] to check entire programs without requiring explicit annotations. While this improves conciseness, it can reduce code readability and produce complex error messages. Many programmers, therefore, choose to write explicit type signatures for clarity, even when inference is available.

To summarize, the type systems of functional programming languages provide rigorous tools for specifying and enforcing program behavior. By encoding system constraints directly in types, they prevent many classes of runtime errors. However, this rigor introduces trade-offs: a steeper learning curve and less flexibility for exploratory or prototype development. In the following sections, we explore how GeckoGraph supports these type-system features by visualizing type-level information that is often implicit or difficult for humans to interpret.

3. GeckoGraph

GeckoGraph is a visual notation for type annotations in statically typed programming languages. It is intended to work tangibly with text-based annotations but uses colors, shapes, and symbols to make structures of types easy to identify at a glance. In this section, we describe the design of GeckoGraph and highlight some unique benefits of programming with GeckoGraph.

3.1. Design of GeckoGraph

The design of GeckoGraph focuses on visualizing types in functional languages (e.g., Haskell, ML). In this paper, we use Haskell as an example. As illustrated in this section, it can express basic types, polymorphic types, algebraic data types, type classes, and a few advanced type-level features. However, GeckoGraph can also be used in imperative and multiparadigm languages such as TypeScript and Rust. We provide a GeckoGraph visualization tool available online for multiple languages [14].

We identified three main design goals for GeckoGraph based on the challenges of using polymorphic types [4, 5] and how programmers tend to use type annotations to support their programming tasks [15], as follows.

(D1) Easy to learn. GeckoGraph should take little to no effort to learn. The rules to translate a text-based type notation to GeckoGraph should be minimal. Where possible, GeckoGraph should be intuitive to programmers who are familiar with text-based type notation.

SYMBOL	DESCRIPTION
A	A type variable a
B	The type Bool
C	The data type $(a\ b)$
D	The data type $(a\ b\ c)$ with two type arguments
E	The data type $(a\ (b\ c))$
F	The function type $a \rightarrow b$
G	The curried function type $a \rightarrow b \rightarrow c$
H	The high order function type $(a \rightarrow b) \rightarrow c$
I	A type with 2 type class constraints $(A\ a,\ B\ a) \Rightarrow a$ For example, $(\text{Ord}\ a,\ \text{Eq}\ a) \Rightarrow a$
J	A data type with with a type class constraint $A\ a \Rightarrow a\ b$ For example, $\text{Functor}\ f \Rightarrow f\ a$

Figure 3: Examples of various types as represented in GeckoGraph, including type variables (A) and concrete types (B). Data types (C, D, E), function types (F, G, H), and type classes (I, J).

(D2) Easy to parse for humans. GeckoGraph should make the task of reading and understanding type notation easy. It should emphasize the less obvious properties of a type signature. GeckoGraph should eliminate the need for mental backtracking, such as counting opening and closing parentheses and remembering which type classes are required on which variables.

(D3) Optimized for comparing and searching. GeckoGraph aims to make the task of comparing two types easy, especially to make subtle differences in text-based notation harder to miss. This also includes the task of choosing an ideal function from a list of potential functions. For example, programmers search for a desired function from a documentation site with only par-

tial knowledge of its type (e.g., the arity of a function, containing a known type, or type class requirements).

Motivated by the design goals, we designed GeckoGraph with the following features and construction rules (Fig. 3):

Simple Types. Simple types, such as type variables and concrete types, are displayed in a cell : a solid-colored rectangular box with an angled corner on the top left. Each type identifier encodes a distinct color hue of the cell. Its first 1 or 2 letters are displayed inside the cell at the bottom left to provide familiarity (design goal 3.1) and a strong secondary encoding. The angled corner in the top left provides visual separation between two cells, even when the same color cells are next to each other, allowing GeckoGraph to be zoomed out to extremely small sizes (Section 6.2.1) without suffering readability (design goal 3.1).

Algebraic Data Types. GeckoGraph displays an algebraic data type as a larger cell, where the type constructor half encloses its arguments: . The arguments are aligned in the bottom right of the cell. Two distinct visual dimensions are used to provide additional visual clarity (design goal 3.1). Data types containing more arguments (e.g., $a \ b \ c$ or $(a \ b) \ c$) will expand horizontally . Data types that are nested (e.g., $a \ (b \ c)$) will grow taller . This distinction accommodates our design goal 3.1. Note that the height of GeckoGraph grows only upwards, but not downwards. Not only does this allow GeckoGraph to be more efficient in its space usage, but it also allows the legend text to be consistently aligned at the bottom and can be read as though it is a text-based type notation (design goal 3.1).

Function Types. Functions are the fundamental building blocks of functional programming languages, and function types are ubiquitous and the most important in type-level programming. In Haskell, (\rightarrow) is defined as an infix type operator with right associativity to provide succinct type annotations. GeckoGraph preserves this syntax feature to make the notation more intuitive (design goal 3.1): the 2 arguments of a function type in GeckoGraph are placed on both sides of the cell . A special function indicator $(>>>)$ is displayed at the top of the cell.

Curried functions are functions that return other functions (e.g., $a \rightarrow (b \rightarrow c)$). In functional languages, they are often written with a syntactic shorthand, such as $a \rightarrow b \rightarrow c$. These two forms are uniquely represented in GeckoGraph as two functions cells merged together . It appears as if the second function covers the right half of the first function, indicating that the second function is the return type of the first. Higher-order functions that take other functions as arguments (e.g., $(a \rightarrow b) \rightarrow c$) follow the rules of functions and nested data types . The placement of function indicators aims to make it easy to find desired functions in the documentation site based on function arity and higher-order functions (design goal 3.1). It is easy to tell high-order functions from the vertical position of the function indicators. Similarly, it is easy to count the arity of a function by counting the number of function indicators at the outer layer. For example, in Fig. 4, the outer layer function indicators (the ones colored in the lightest shade of gray) suggest a ternary function. The two argument types and the return type are all functions.

Type Classes. Type classes are an intrinsic part of Haskell [9], and many other functional languages. In GeckoGraph, the type classes (e.g., $(A \ a, B \ a) \Rightarrow a$) are indicated in the extended area below one or more GeckoGraph cells . Each type class required on a type variable displays as a square indicator aligned on the right of the extended area. In GeckoGraph, a

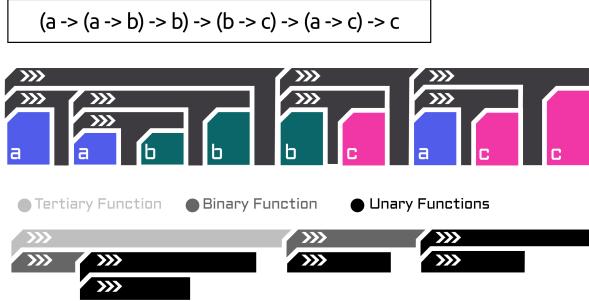


Figure 4: An example of the visual hierarchy of the function indicator. The function indicator can be used to easily identify the arity of a function type by counting the connecting function indicators.

type-class glyph is embedded in every type variable that requires this class. This means when displaying the type $(==) :: \text{Eq } a \Rightarrow a \rightarrow a \rightarrow \text{Bool}$ in GeckoGraph, the constraint `Eq` appears in twice on both occurrences of the type variable `a`.

The GeckoGraph type class's design promotes the type class placements rather than the type class names. Programmers can easily see where and how many type classes are required, but they may need an extra step (global legends of the color mapping or a pop-up window) to identify the name of the type class. We believe that this is a well-justified trade-off. The strength of this approach lies in situations when the type class context is complex. When reading a type $(A \ a, A \ c, B \ a, B \ b, C \ b) \Rightarrow a \rightarrow b \rightarrow c$, programmers may need to switch back and forth to remember which type classes are needed on which variable. In GeckoGraph, type classes are explicitly displayed under the type variable cell, minimizing programmers' effort to associate each type variable with all its type classes (design goal 3.1).

3.2. Key Design Criteria of GeckoGraph

GeckoGraph was designed through many different iterations. Many research methods were used to verify ideas, including prototyping, cognitive walk-throughs, and formative studies. We discuss some notable visual dimensions in GeckoGraph and their justifications.

Visual Dimensions of GeckoGraph. **The colors of GeckoGraph** help programmers to see the permutation of type variables in the input and output of a function. A recent review [16] of 59 graphical perception articles showed that combining solid color hue in a filled shape provides stronger visual perception for nominal data such as type identifiers. GeckoGraph uses a qualitative color scheme from ColorBrewer [17] to encode symbolic values of types. One example of GeckoGraph's use of color is the "rotation" function (Fig. 5) in the user study (Section 4). With text-based type notation, programmers often rely on mnemonic devices such as alphabetic ordering or naming conventions. For example, the rotation function $f2 :: \text{Zero } a \ b \ c \ d \rightarrow \text{Zero } b \ c \ d \ a$ appeared in the evaluation (Section 4), shuffles the type arguments one step to the left. The purpose of this function becomes less recognizable if changed to $f2 :: \text{Zero } e \ v \ m \ h \rightarrow \text{Zero } v \ m \ h \ e$.

The horizontal axis of GeckoGraph is often helpful for identifying differences in function arities. For example, in Fig. 6, the programmer intended to implement a function that sums 3 integers. In the implementation, the programmer missed a `(+)` function at the end; the resulting

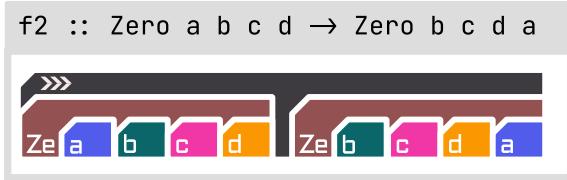


Figure 5: The ‘f2’ function in the user study. It shuffles the type arguments one step to the left, and the leftmost argument wraps around to be the rightmost.

function type is largely different in length. It is also clear that the function needs to apply to one more binary function to satisfy the length requirement.

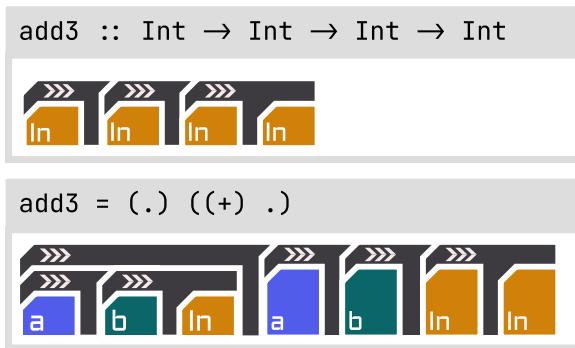


Figure 6: An implementation of function `add3` but the author missed an `(+)` from the correct implementation `(.) ((+) .)`. GeckoGraph highlights the difference in arity and reveals that a binary function is needed on the right-hand side for the arity to match.

The vertical axis of GeckoGraph often helps identify the number of levels in a nested complex type. This can often be very useful when inspecting mismatched types where data types are nested at different levels. Common examples include when programmers forget to apply the value to “return” in a monadic block or to use `liftIO` to cast an `IO` effect. In Fig. 7, the uses of `return` are excessive. It can be easily identified by examining the difference in the vertical layers of the two types. In text-based type notation, this is distinguished by different pairs of parentheses. However, parenthesis is an overloaded syntax in type notation. In Haskell, parentheses are used to enclose tuples `(a, b)`, specify the fixity `(a -> b) -> c`, or have no effect `a -> (b -> c)`.

Support Advanced Type Features. The design of GeckoGraph enables the visualization of many advanced type-level features. **Kind visualization:** The kind system [18] is used in Haskell to indicate whether a type is a type constructor (e.g., `[]`, `Maybe`) or a type constant (e.g., `Bool`, `Char`). Because of this, kind is often described as the type of types. GeckoGraph infers the *kind* of type variables and consistently displays this information. For example, in Fig. 8 (A), the variable `a` is inferred to have the kind `* -> *` because of its use on the right-hand side. GeckoGraph respects this *kind* information and displays the same kind level on the first occurrence of `a`, but over an empty structure, indicated using a dotted outline. **Qualified constraints:** GeckoGraph’s type

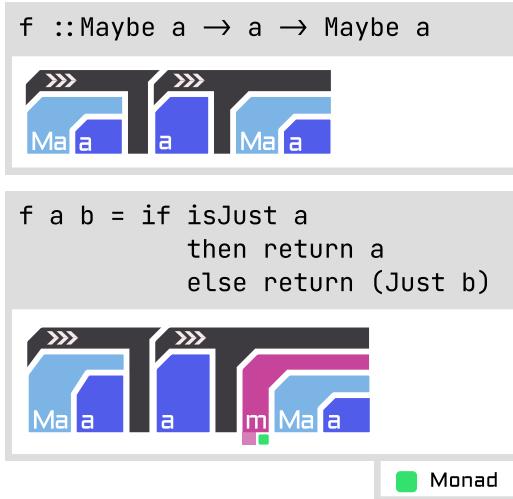


Figure 7: The function `f` is planned to have the type `Maybe a -> a -> Maybe a`. The programmer mistakenly applied the result to the `return` function, making the result inside a Monad instance. GeckoGraph reveals the difference in the “layers” of types.

class notation naturally extends to support qualified constraints. In the type `forall b. A b -> a b`, GeckoGraph shows the scope type class requirement on `a b` (Fig. 8 B). **Multiple Parameter Type Class:** GeckoGraph supports multiple parameter type classes by using multiple shapes with the same color hue to indicate the different parameters of the same type class. For example, for the type `A a b -> a b`, GeckoGraph shows that the variables `a` and `b` both need an `A` class, but they are the different parameters of `A` (Fig. 8 C).

Provide Precise Interactivity. Modern programming environments often allow programmers to mouse over part of the source code to query detailed information, such as definition, references, or documentation. However, with text-based source code, it is often hard to distinguish whether programmers want the most specific fragment under the cursor or larger blocks. Because of its graphical layout, GeckoGraph allows programmers to precisely select which part of a type signature they intend to query, that is, in Fig. 8 (D) when the user mouses over the type class box (orange square) under the second occurrence of `a` the type class it represents is revealed in detail, a specific query that is hard to execute with point and click over text based type annotation.

4. Evaluation

To evaluate the effectiveness of GeckoGraph, we designed and conducted a gamified controlled experiment. The study, “Zero to Hero”, adopted a game-like experience. It contains 10 tasks (levels) of varying difficulty. At each level, participants are asked to implement a function named “`zeroToHero`” with the help of a limited list of provided functions. These functions are different at each level, and the target types of Zero and Hero vary at each level as well. The details of each level are provided in the Appendix Appendix A.

The experiment aims to study how programmers use and reason about polymorphic types when performing programming tasks. In particular, we studied how programmers scan and select

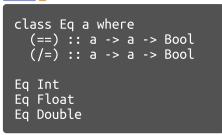
SYMBOL	DESCRIPTION
A 	Type $(A\ a\ b \rightarrow a\ b)$. The kind of a is at least $* \rightarrow *$
B 	$(\text{forall } b. A (a\ b)) \Rightarrow a\ b$ (quantified constraints)
C 	$A\ a\ b \Rightarrow a\ b$ (Multiple Parameter Type Class)
D   <pre> class Eq a where (==) :: a -> a -> Bool (/=) :: a -> a -> Bool Eq Int Eq Float Eq Double </pre>	Precise sub-expression selection of a type signature

Figure 8: Advanced features of GeckoGraph. (A) Kind visualization. (B) Qualified constraints. (C) Multiple Parameter Type Classes. (D) Precise selection of sub-structures.

potentially useful functions from a library and how they compare intended types and actual types when type errors are encountered.

4.1. Example Levels

We illustrate the tasks of the user study using levels 2 and 3 as examples. For level 2 (Fig. 9), the players need to implement the function `zeroToHero :: Zero a -> Hero b`. The available functions are `runZero :: Zero a -> a`, `makeHero :: a -> Hero a`. A generic function (\$) is provided to improve the ergonomics of composing functions, but all tasks can be completed without the use of generic functions. The possible solutions and other details of the level can be found in Appendix Appendix A. To complete this level, players only need to use each of the provided functions once. One possible solution to this task would be `zeroToHero z = makeHero (runZero z)`. The user illustrated in Fig. 9 did not reach a correct solution, as the type signature and GeckoGraph in *To Implement* panel do not match the one in *Your Result* panel.

In level 3 (Fig. 10), players need to implement the function `zeroToHero :: Zero a -> Hero (a, a)`. In this level, not all provided functions are useful. In particular, players must discover that only `f1` and `f3` are necessary to produce the desired results. One implementation that satisfies the target type is `zeroToHero z = f3 (f1 z)`. Another possible implementation is `zeroToHero z = f3 . f1 $ z`, exercising the Haskell function composition idioms. Again, the user illustrated in Fig. 10 did not reach a correct solution, as the type signature and GeckoGraphs in *To Implement* and *Your Result* do not match.

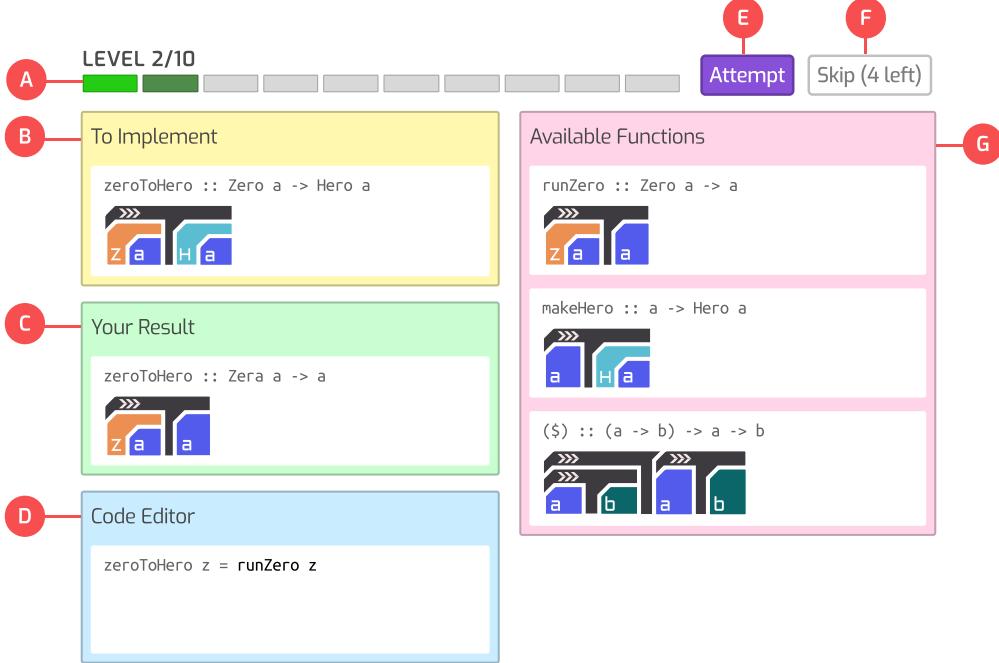


Figure 9: A screenshot from the user study. For this task – level 2 (Shown at A) – the players need to implement the function `zeroToHero :: Zero a -> Hero a` (B). They write their own definitions in the code editor (D) using a set of provided functions (G). The inferred type of their current definition is shown in (C). When ready, they can test their solution by clicking on the *Attempt* button (E). They can also skip a level by clicking on the *Skip (n left)* button next to it (F).

4.2. Recruitment

Participants were recruited online through the Haskell community on Reddit and Discord. Participation is fully anonymized; detailed ethical implications of these experiments were reviewed and approved by the IRB of the authors’ institution.

4.3. Group Assignments

Participants are assigned to one of two groups (A and B). The experiment uses between-subjects design [19]. The tasks assigned to the two groups are the same. All participants receive both treatments (with and without GeckoGraph) during their runs, but in a different order. **Group A** is assisted by GeckoGraph on even levels and text-based type annotation on odd levels. *Group B* is assisted by text-based type annotation on even levels and GeckoGraph on odd levels. The number of participants in the two groups is counterbalanced.

4.4. Hypothesis

In programming tasks that involve reading and understanding polymorphic types, graphic notation using visual elements that provide higher grouping strength (colors, shapes, sizes, and symbols) can improve the performance of such tasks compared to traditional text-based type notation. Our null hypothesis is: *Using graphic notation has no effect compared to traditional*



Figure 10: A screenshot showing level 3 of the user study. In this level, the players need to implement the function `zeroToHero :: Zero a -> Hero (a, a)`. Players are helped by the functions: `f1 :: Zero a -> Hero a`, `f2 :: Zero a -> (a, a)`, and `f3 :: Hero a -> Hero (a, a)`.

text-based type notation. This hypothesis and the task design were registered at the Open Science Foundation prior to data collection [20].

4.5. Task Design

During the study, participants received 10 tasks. The tasks start off easy but gradually increase in difficulty. In each task, a target type of the function `zeroToHero` is shown to the participants. Participants are provided with a list of available functions to implement the target function and to ensure it assumes the target type. This is to simulate the tasks of selecting useful functions from a library. Participants are not allowed to use any other functions or variables outside the provided functions; even the Haskell prelude is not available. This ensures that everyone has the same knowledge and minimizes the effect of familiarity.

Participants can skip a level if they are stuck. We believe that it is normal for anyone to get stuck on a challenging task, and being stuck on one of the 10 tasks does not discount their

feedback on the tool. The number of skips a participant can use is limited to four, so that it is not possible to submit qualitative feedback without completing at least 6 tasks out of 10.

4.6. Measurements

During the study, the time spent by participants on each task is recorded. We also record the resulting status of each level, whether it is a success or a failure. Before starting the study, participants nominate their level of Haskell experience on a four-level scale: beginner, familiar, knowledgeable, and expert. If a participant has completed all 10 levels (with the help of skipping), we invite the participant to complete a post-study survey. In it, we ask for them to rate how **intuitive** the GeckoGraph design is, how **distracting** they find GeckoGraph, and how **helpful** GeckoGraph is, using a seven-point scale. In the end, we ask 2 open-ended questions: *What is your overall experience using GeckoGraph* and *What are your expectations on the potential applications of GeckoGraph*.

Data collection of the study was stopped after the planned cut-off period of 14 days. After the cut-off date, the ZeroToHero website is open-sourced and available for free evaluation¹, and repeating our experiment. However, no further data was collected.

5. Results

During the data collection period, a total of 714 users participated in the study. Among them, 245 are novice users, 216 are familiar with Haskell, 216 are knowledgeable users, and 88 are expert users.

5.1. Time to complete levels

The 10 levels are designed to increase difficulty gradually. From the results of the experiment, most of the tasks align with this trend. However, three tasks stand out in Fig. 11. Level 7 (mean = 334 seconds) is the hardest task in terms of time, followed by level 8 (mean = 228 seconds) and level 5 (mean = 224 seconds). To complete an average level, the beginner group uses an average of 100 seconds, the familiar group uses 90 seconds, the knowledgeable group uses 80 seconds, and the expert group uses 70 seconds. This roughly aligns with self-reported expertise. We show that the task time on each level follows normal distributions using a Shapiro-Wilk test [21] ($p\text{-value} \leq 1.018 \times 10^{-16}$, for an alpha value of 0.05, p less than 0.05 is considered a normal distribution).

Levels 5, 7, and 8 are the only three levels that include functions from the standard Haskell library other than the generic `(.)` and `(\$)`, which are provided for pure convenience. Level 5 requires programmers to use the `fst` and `snd` functions to extract a value from a tuple. Level 7 requires programmers to use the `(<*>)` function of the `Applicative` class, while level 8 requires the `fmap` function of the `Functor` class. The authors speculate that the more experienced participants are much more familiar with these functions, hence the strong contrast on these three levels.

However, when comparing the task time between the two treatments, we were unable to reject the null hypothesis. In a two-sample T-test, we could not find any significant difference between the two groups overall ($p\text{-value} = 0.457$), nor did there differ between the two groups in any of the four levels of experience (beginner: $p\text{-value} = 0.845$, familiar: $p\text{-value} = 0.524$, Knowledgeable: $p\text{-value} = 0.712$, expert: $p\text{-value} = 0.771$).

¹<https://zerotohero.fly.dev>

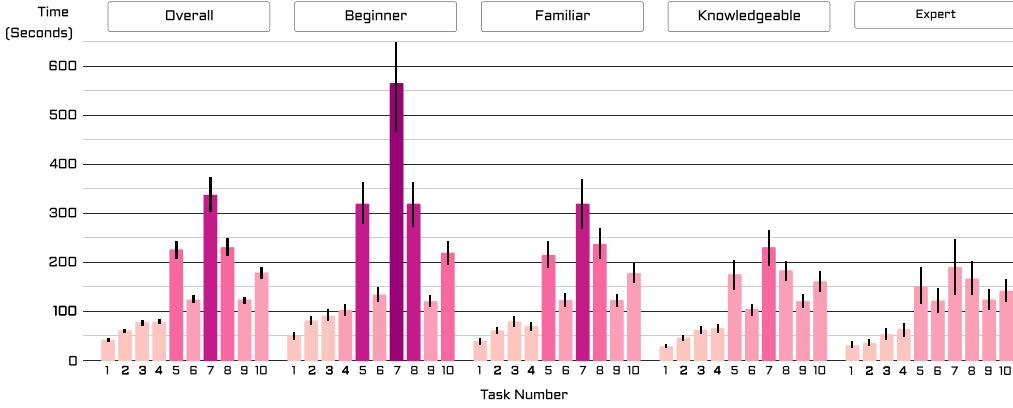


Figure 11: Time spent on each level, with 95% confidence interval. We show that the difficulty steadily increases across the user study, but levels 5, 7, and 8 are significantly harder than the authors intended. The overall task time of each group roughly matches the experience level.

5.2. Success rate

We saw that, overall, GeckoGraph provides a slightly higher success rate (96.88%) than text-based type notation (94.62%). This trend can be seen in every experienced group: beginner group (95.12% vs. 92.68%), familiar group (97.39% vs. 93.34%), knowledgeable group (96.82% vs. 96.06%), and expert group (98.2% vs. 96.40%). We saw the significance decrease as the user's experience increased. When performing a proportion test on each group, we see that the effect is most significant with the beginner group and reject the null hypothesis (z score = 2.0228, p -value = 0.0431), followed by the familiar group (z score = 1.7495, p -value = 0.0802). The knowledgeable group (z score = 1.0295, p -value = 0.3032) and the expert group (z score = 0.8660, p -value = 0.3756) show less significant differences between treatments.

When breaking down the result in each task (Fig. 12), we were able to reject the null hypothesis in task 10 of the beginner group (94.9% with GeckoGraph, and 80.4% without GeckoGraph, p -value=0.0452) and task 10 of the familiar group (98.5% with GeckoGraph, and 87.3% without GeckoGraph, p -value = 0.136).

Level 10 is the most cognitively demanding: the provided functions are the lengthiest, and the solution is challenging and deceptive. Programmers must use the provided rotate function $f2$ to adjust the positions of the four components of the *Zero* value four times to get to the correct position. Completing the level requires applying the functions $f1$, $f2$, and $f3$ in a nested sequence that is 8 layers deep. Consequently, programmers need to frequently switch contexts to recall the purpose of each function. We speculate that this level demands the mental flexibility typical of a Haskell expert to internalize these functions quickly. GeckoGraph helps less experienced programmers to achieve the same level of comprehension fluency by adding visual clarity.

5.3. Qualitative Feedback

In their responses to the post-study survey, participants showed mostly neutral sentiments towards GeckoGraph's design and appeal. They find GeckoGraph moderately intuitive to use and that its appearance in the interface does not cause distraction.

For the question "Do you find the GeckoGraph distracting", most of the participants rated a low score, with an average of 2.88 (Fig. 13 left). For the question "How intuitive do you

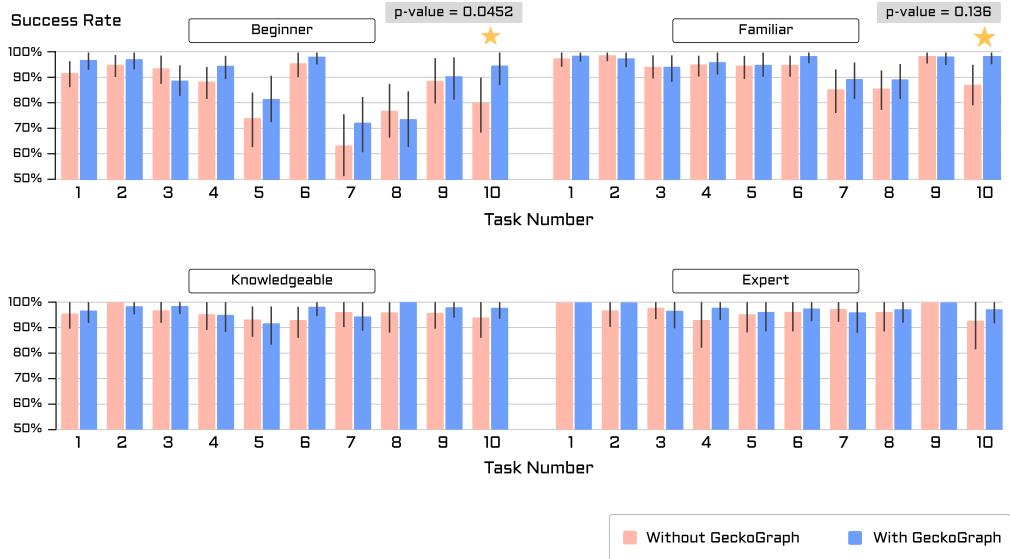


Figure 12: **Success rate of each task, with 95% confidence interval, grouped by experience level.** The figure is cropped from 50% to 100% for readability. Bootstrapping was used for computing the confidence interval because of the non-normal distribution. In most tasks, GeckoGraph provides no significant differences. However, substantial improvements were observed in task 10 for the beginner group (14.5%) and the familiar group (11.2%).

find the GeckoGraph?", we saw a reverse correlation of experience (Fig. 13 Middle): experts find the GeckoGraph most intuitive (5.07), followed by the knowledgeable group (4.87), and the familiar group (4.80). The beginner group found it to be the least intuitive but still rated it positively, albeit with a low score of 4.71. When answering the question "How helpful do you find GeckoGraph in finding the solution?", the answer is more divided into different experience groups (Fig. 13 right). It is slightly positive for beginners (mean=4.25) and slightly negative for the other groups, the familiar group (mean=3.86) and the knowledgeable group (mean=3.32). The expert group finds that GeckoGraph is moderately unhelpful (mean=2.95).

Two interesting trends emerge. First, experienced users are more likely to find GeckoGraph intuitive and easy to use. This aligns with our design decision to make GeckoGraph resemble text-based type annotation. Those familiar with text-based type annotation find GeckoGraph second nature to read. Second, the correlation reverses when it comes to perceived helpfulness. The more experienced the users are, the less likely they are to find GeckoGraph helpful. We speculate this pattern occurs because users who are experts in text-based type annotations naturally prefer reading text-based notation and tend to ignore GeckoGraph, diminishing its usefulness. Conversely, novice users are more likely to be drawn to GeckoGraph and find that its visualization offers more clarity and useful insights.

5.4. Threats to validity

Task design. In our user study, most of the provided functions are very abstract. These functions are created by the authors solely for the gamified study. They are designed to be different from well-known Haskell functions to minimize the familiarity variable. They are also designed with

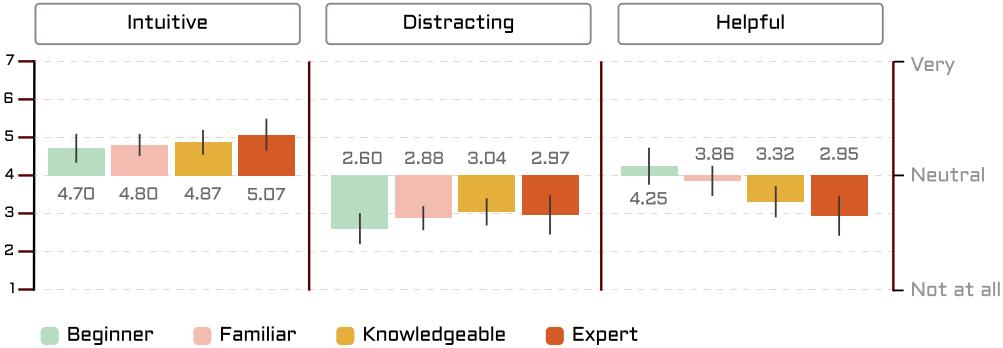


Figure 13: The users rated scores of how intuitive (left), distracting (middle), and helpful (right) they found GeckoGraph, with 95% confidence interval. Overall, programmers are neutral to GeckoGraph’s design. Some find GeckoGraph to be intuitive and not distracting. However, beginners are more likely than experienced functional programmers to find it helpful.

an interest in being challenging and fun. These functions may not be the most representative of real-world Haskell programming.

The use of skips. Although we justified the use of skips in Section 4.5, the availability of skipping does allow users to adopt more utilitarian strategies, often involving skipping a level without giving it a fair try. This happened more often in the later levels when users realized they had enough skip opportunities left to “complete the game”. These strategies may result in lower recorded success rates than if no skips were allowed.

6. Discussion

This section discusses the main findings and implications of our study. We speculate the strengths of GeckoGraph observed from both quantitative and qualitative results (Section 6.1). We then discuss the limitations of the current design (Section 6.2) and reflect on the gamified nature of our user study (Section 6.3). Finally, we identify potential applications of GeckoGraph in programming tools, documentation, and education (Section 6.4).

6.1. Strengths

Overall, our experimental results show no statistically significant difference in task completion time between conditions with and without GeckoGraph. However, we find that using GeckoGraph shows a significant, albeit moderate, effect on task success rates for beginners. In particular, on the hardest task (task 10), GeckoGraph improved the success rates across the two less experienced user groups by 14.5% and 11.2%, respectively. To interpret the observed advantages, we hypothesize the practical benefits that GeckoGraph offers to programmers.

6.1.1. Identifying the Most Important Features

Qualitative feedback revealed that programmers found GeckoGraph particularly helpful for spotting patterns and identifying important features in type signatures. Participants appreciated how GeckoGraph highlights key aspects of a type using distinctive visual properties such as color,

length, and height. Representative comments include: “*Types are much easier to understand with GeckoGraph than by trying to parse all parentheses and understand the types from the signature.*” and “*It makes it easier to see at a glance when your output type is correct or what the difference between the current type and the target is.*”

6.1.2. Generalizing Patterns in Type Classes

A common source of confusion among novice Haskell programmers is distinguishing between functions that operate on generic types and those that work on concrete types [22]. For example, differentiating between `Integral a => a` and `Integer`, `Floating a => a` and `Float`, or `Foldable` and `[]` can be difficult. Learners often find such distinctions irrelevant to their immediate tasks, which complicates problem-solving and error interpretation.

As shown in Fig. 1, when a programmer mistakenly supplies a `Char` instead of a `String`, the compiler reports a confusing mismatch involving a generalized type `t0 a0`. In GeckoGraph, both list types and types with a `Foldable` instance share similar visual structures. This correspondence helps programmers relate generalized types back to familiar concrete ones, lowering the cognitive barrier to understanding polymorphism. Such visual consistency is especially valuable in Haskell, where even simple arithmetic operators like `+` and `-` are defined generically.

6.1.3. Consistent Color Scheme

A frequent task in programming is scanning through documentation or libraries to find a desired function, often based on partial knowledge such as argument types, arity, or expected output. For example, one might look for a conversion from `String` to `Data.Text`, or a function like `lookup` that operates on `Data.Map a b` and returns a value of type `b`. GeckoGraph supports this by assigning consistent colors to identical type identifiers, allowing programmers to visually group related types and quickly locate relevant functions.

6.1.4. Low Barrier to Learning and Understanding

GeckoGraph retains key conventions from traditional textual type notation: it follows left-to-right reading order, preserves symbolic names as secondary encodings, and mimics prefix and infix structures. These design choices ensure that programmers familiar with Haskell can intuitively understand and use GeckoGraph with minimal training.

This ease of adoption was evident in user feedback. Several participants noted that they had no prior Haskell experience but were able to solve the puzzles successfully with GeckoGraph’s help. Comments included: “*It is similar enough to traditional types that it feels intuitive.*” and “*This was how I parse the textual representation of types.*”

6.1.5. Language Agnostic

GeckoGraph can be implemented in any statically typed language. This makes it useful in multi-language programming environments—such as projects combining client-server architectures or foreign function interfaces—where it can provide a unified visual notation across different syntaxes. In education, GeckoGraph abstracts away naming differences between languages. For example, the product type in algebraic data types is known variously as a *record*, *struct*, or *named tuple*. GeckoGraph helps unify these equivalent concepts visually, promoting language-independent understanding.



Figure 14: **Envisioning GeckoGraph as programming assistance.** (Left) Display GeckoGraph as type hints; (Right) Visualize mismatched types with GeckoGraph.

6.2. Weaknesses

6.2.1. Space Usage

GeckoGraph’s layout scales horizontally with the size of a type’s syntax tree and vertically with its depth. Compared to traditional text-based notation, this results in higher vertical space consumption. We have identified possible optimizations, such as displaying only color blocks while omitting textual identifiers, to reduce space usage while preserving the main advantages of the design.

6.2.2. Color Encoding

GeckoGraph relies heavily on color as its primary visual channel. While this offers strong perceptual grouping [16], color perception varies among individuals, posing accessibility challenges for visually impaired programmers. Although GeckoGraph adopts color-blind-friendly palettes, this only mitigates some issues. To improve accessibility, future work will explore alternative encodings such as patterns and shapes in addition to color.

6.3. Gamified Human Study

Our user study was intentionally designed as a game-like experience, incorporating multiple gamification techniques such as levels, story elements, and goals/rewards [23]. This approach ensured participant motivation and contributed to a historically high participation rate within the Haskell community. Gamification has been widely shown to enhance engagement and motivation in human–computer interaction research [24]. In our context, it helped attract a diverse set of participants across a wide range of experience levels, addressing a common challenge in studies of functional programming.

6.4. Potential Applications

6.4.1. Programming Assistance

Post-study feedback highlighted a strong interest in integrating GeckoGraph into programming environments. Modern IDEs already allow users to inspect expression types via hover tooltips; GeckoGraph could enhance this feature by visualizing type structures directly in the tooltip, as envisioned in Fig. 14 (Left). Similarly, GeckoGraph could be used to visualize mismatched parts of conflicting types during compilation errors, as illustrated in Fig. 14 (Right).

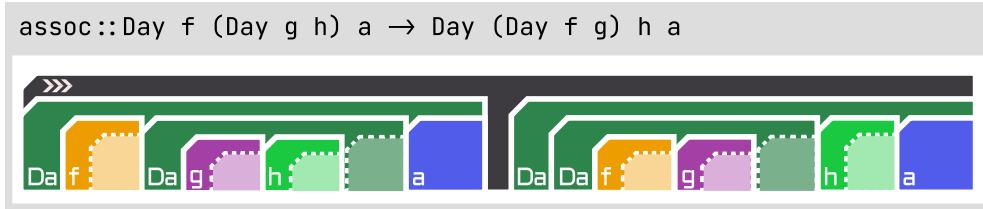


Figure 15: The `assoc` function for day convolution [26] in the Kan extension. Even without knowing the exact definitions, viewers can easily see that `f`, `g`, and `h` are all higher-kinded types—information that is obscured in traditional textual notation.

6.4.2. Documentation Assistance

Our findings suggest GeckoGraph would be highly effective for visualizing function types in API documentation and programming libraries. Because GeckoGraph representations can be generated automatically, integration into documentation systems like Hoogle [25] would require minimal effort. A quick visual scan of the page can quickly review the theme of a library or package, i.e., the domains and codomains most functions operate on. For instance, a library that mostly supports string operations should predominantly show the color of `String` type, while a low-level graphics-related library may show mainly the colors for `Vector` type and `Float` type.

6.4.3. Pedagogical Applications

GeckoGraph can serve as a valuable teaching aid for introducing programming language concepts that are abstract or difficult to explain textually. Several study participants with no prior Haskell experience reported that GeckoGraph helped them understand type-related puzzles and complete all tasks successfully.

Beyond basic learning, GeckoGraph’s advanced features (Section 3.2) also make it suitable for teaching higher-level functional programming concepts. For instance, consider the `assoc` function for day convolution [26] in the Kan extension (Fig. 15). Although the type signature is concise, it is cognitively difficult to parse due to multiple variables and their hidden kinds. GeckoGraph reveals these higher-kinded types visually, exposing partially applied data types and making the semantics easier to understand.

7. Related work

This section reviews prior work in three areas relevant to this research. First, we examine approaches to the visualization of types in programming languages. We then broaden our view to general visualization techniques applied to programming environments, including topics like memory allocation and change history. Finally, we review studies comparing visual and textual representations in programming education and environments.

7.1. Visualization of Types

A similar half-enclosing structure was proposed in Jung’s visualization of types [27]. In Jung’s notation, the type constructor partially encloses its arguments, but its figure is positioned at the bottom right (Fig. 16). In contrast, GeckoGraph follows a natural left-to-right reading order, allowing larger structures in a type signature to take visual precedence over smaller ones.

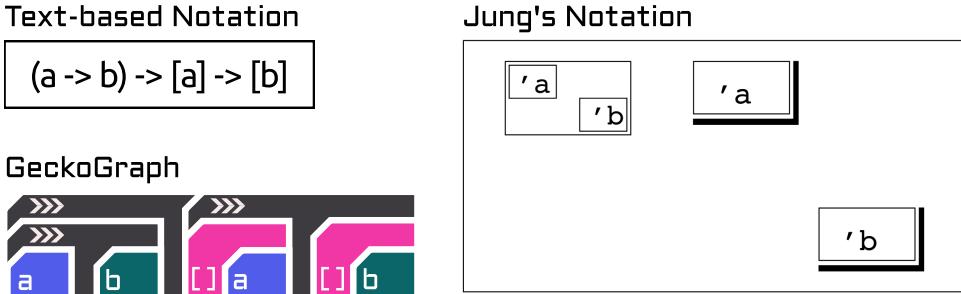


Figure 16: Comparison of the `map` function $((a \rightarrow b) \rightarrow [a] \rightarrow [b])$ in text notation, GeckoGraph, and Jung’s notation.

Compared to Jung’s approach, GeckoGraph offers two major advantages. First, it naturally represents the general notion of a curried function. Partial application can be intuitively read as removing the first argument, which is not as clear in Jung’s notation. Second, the shape of a function type remains visually consistent with that of normal data types. In Jung’s design, a function type such as $a \rightarrow b$ looks markedly different from a data type $f\ a\ b$. This distinction is important because, in functional languages, abstractions often generalize over both functions and data types. For example, both functions and lists have Functor instances, allowing their inner values to be transformed using `fmap`. GeckoGraph’s consistent visual shape helps make such generalizations more apparent.

More recent work, TYPEical [28], provides an interactive visualization of function types for exploring and comparing functions in R packages. This project is innovative in that it uses a Sankey diagram [29] to illustrate the flow between collections of functions and the data types they operate on, as well as the frequency of their use, providing a high-level overview of the usage statistics. GeckoGraph shares TYPEical’s vision that visualization should support three key tasks: **finding functions**, **determining types**, and **comparing types**. However, unlike GeckoGraph, TYPEical’s visualization is generated ahead of time and is aimed primarily at language designers and core library developers rather than students or everyday programmers.

7.2. Visualization in Programming

The use of visualization to improve the comprehension of polymorphic types is not new. It has been applied to represent document structures, runtime behavior, and static analysis results. For example, FluidEdt [30] displays heap graphs in the editor margin, while I3 [31] presents search similarity and change history through compact block-based diagrams. Almeida et al. [32] introduced a visualization technique to aid understanding of ownership and borrowing in Rust. Although graphical representations of types remain a relatively narrow field, it has been explored in research such as Clerici et al. [33], who proposed a graph-based type inference system that visualizes unification states.

GeckoGraph positions itself similarly to these projects, using color, shapes, and icons to provide easily interpretable visual cues. However, GeckoGraph focuses specifically on type-level information—an area largely unexplored by previous work. In addition, GeckoGraph was

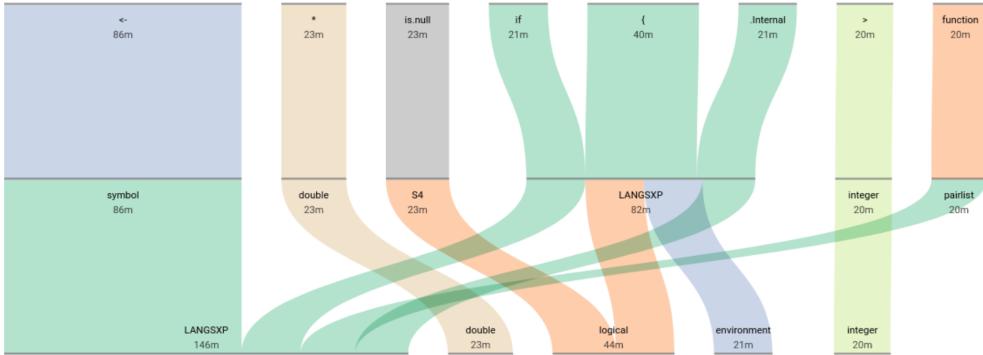


Figure 17: TYPEical’s type flow visualization. The figure displays the type signatures of functions in R’s `base` library (top axis) and their first and second arguments (middle and bottom axes).

evaluated through a larger and more diverse study, including participants with a broader range of programming experience.

7.3. Visual vs. Textual Representation

Many studies have compared visual programming environments with textual ones. Research [34, 35, 36, 37] consistently shows that students taught with visual programming languages demonstrate greater confidence, retention, and enjoyment compared to those using purely textual languages. While following a similar trend, GeckoGraph is designed as a complement to text-based notation rather than a replacement for it.

Several tools have explored the effect of visual augmentation—adding visual representations alongside textual code. For example, Greenfoot [38] integrates visual and textual programming elements to support learning, while PILeT [39] adapts its presentation style to users’ preferences. Both systems report positive effects of visual augmentation. Although GeckoGraph also combines visual and textual representations, it differs in that prior studies focused on imperative languages such as Java and Python, whereas our evaluation investigates its effect in a functional programming context (Haskell).

8. Conclusion and Future Work

In this paper, we propose GeckoGraph, a graphical notation for type annotations in functional programming languages. GeckoGraph aims to complement traditional text-based type notation by making reading, understanding, and comparing types easier. We conducted a large-scale human study using GeckoGraph, comparing it to text-based type notation. This is the largest user study on functional programming we are aware of. Although no significant effect was found on the overall time to complete each programming task, we found a modest effect on the success rate in the beginner group. In one of the more difficult programming tasks, we find that GeckoGraph significantly improves the success rate by 14.5% for beginners and 11.2% for less experienced programmers.

Our work in this area opens many new directions for future research. In particular:

In-the-wild Studies Although our experiment scenarios are drawn from real-world programming tasks, a certain level of variable control is still applied to remove the effect of familiarity with the

tools and libraries. However, it is necessary to assess the usefulness of tools such as GeckoGraph in terms of their real-life usage. Different research methods should be used to study the effects of GeckoGraph on realistic programming tasks. This may include field deployments or case studies. **Alternative Type Visualization** We strongly believe that visualization is an underutilized technique in this effort. GeckoGraph focuses on a faithful view of the tree structure of programming types. However, many more areas and types demand a visual approach. For instance, visualizing the ordinal relationship of subsumption or visualizing the numeric values in dependently typed “type programs”.

References

- [1] L. Cardelli, Basic polymorphic typechecking, *Sci. Comput. Program.* 8 (2) (1987) 147–172. doi:10.1016/0167-6423(87)90019-0.
- [2] S. Klabnik, C. Nichols, Generic data types, <https://doc.rust-lang.org/book/ch10-01-syntax.html>, accessed: 2024-2-20.
- [3] R. Griesemer, I. L. Taylor, An introduction to generics - the go programming language, <https://go.dev/blog/intro-generics>, accessed: 2024-2-19.
- [4] Y. Jun, G. Michaelson, P. Trinder, How do people check polymorphic types?, in: PPIG, academia.edu, Cozenza Italy, 2000, p. 6.
- [5] Y. Jun, G. Michaelson, P. Trinder, Helping students understand polymorphic type errors, in: 8th Annual Conference on the Teaching of Computing, 2000.
- [6] J. Hage, Solved and open problems in type error diagnosis?, *CEUR Workshop Proc.* 2707 (2020) 62–74.
- [7] R. Milner, A theory of type polymorphism in programming, *J. Comput. Syst. Sci.* 17 (3) (1978) 348–375. doi:10.1016/0022-0000(78)90014-4.
- [8] D. MacQueen, R. Harper, J. Reppy, The history of standard ML, *Proc. ACM Program. Lang.* 4 (HOPL) (2020) 1–100. doi:10.1145/3386336.
- [9] P. Hudak, J. Hughes, S. Peyton Jones, P. Wadler, A history of haskell: being lazy with class, in: Proceedings of the third ACM SIGPLAN conference on History of programming languages, HOPL III, ACM, New York, NY, USA, 2007, pp. 12–1–12–55. doi:10.1145/1238844.1238856.
- [10] M. A. AbdelGawad, An overview of nominal-typing versus structural-typing in OOP, *arXiv [cs.PL]* (Sep. 2013). arXiv:1309.2348.
- [11] R. Milner, A theory of type polymorphism in programming, *J. Comput. Syst. Sci.* 17 (1978) 348–375.
- [12] Java Developers, Java - local variable type inference, <https://docs.oracle.com/en/java/>, accessed: 2024-4-16 (2023).

- [13] L. Damas, R. Milner, Principal type-schemes for functional programs, in: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '82, POPL '82, ACM Press, New York, New York, USA, 1982, pp. 207–212. doi:10.1145/582153.582176.
- [14] S. Fu, GeckoGraph: A visual language for types, <https://gecko.typecheck.me/>, accessed: 2025-4-6 (Apr. 2025).
- [15] Justin Lubin, How statically-typed functional programmers author code, in: Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems, Association for Computing Machinery, New York, NY, USA, 2021, pp. 1–6. doi:10.1145/3411763.3451515.
- [16] Z. Zeng, L. Battle, A review and collation of graphical perception knowledge for visualization recommendation, in: Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems, no. Article 820 in CHI '23, Association for Computing Machinery, 2023, pp. 1–16. doi:10.1145/3544548.3581349.
- [17] M. Harrower, C. A. Brewer, ColorBrewer.org: An online tool for selecting colour schemes for maps, *Cartogr. J.* 40 (1) (2003) 27–37. doi:10.1179/000870403235002042.
- [18] M. P. Jones, A system of constructor classes: overloading and implicit higher-order polymorphism, *J. Funct. Programming* 5 (1) (1995) 1–35. doi:10.1017/S0956796800001210.
- [19] R. A. Fisher, The design of experiments, 5th ed, *Ther. Hung.* 5 (1949) 242.
- [20] P. J. Stuckey, T. Dwyer, S. Fu, GeckoGraph: Diagrammatic notation for haskell types (Dec. 2023). doi:10.17605/OSF.IO/TWB4A.
- [21] S. Shaphiro, M. Wilk, An analysis of variance test for normality, *Biometrika* (1965).
- [22] V. Tirronen, S. Uusi-Mäkelä, V. Isomöttönen, Understanding beginners' mistakes with haskell, *J. Funct. Programming* 25 (2015) e11. doi:10.1017/S0956796815000179.
- [23] J. Hamari, J. Koivisto, H. Sarsa, Does gamification work? – a literature review of empirical studies on gamification, in: 2014 47th Hawaii International Conference on System Sciences, IEEE, 2014, pp. 3025–3034. doi:10.1109/HICSS.2014.377.
- [24] J. He, M. Bron, L. Azzopardi, A. de Vries, Studying user browsing behavior through gamified search tasks, in: Proceedings of the First International Workshop on Gamification for Information Retrieval, GamifIR '14, Association for Computing Machinery, 2014, pp. 49–52. doi:10.1145/2594776.2594787.
- [25] N. Mitchell, Hoogle, <https://hoogle.haskell.org/>, accessed: 2024-2-19.
- [26] B. Day, On closed categories of functors, *Lecture Notes in Mathematics* 137 (1970) 1–38.
- [27] Y. Jung, G. Michaelson, A visualisation of polymorphic type checking, *J. Funct. Programming* 10 (1) (2000) 57–75. doi:10.1017/S0956796899003597.
- [28] C. Moy, J. Belyakova, A. Turcotte, S. Di Bartolomeo, C. Dunne, Just TYPEical: Visualizing common function type signatures in R (Aug. 2020).

- [29] T. Keiderling, Der Brockhaus, Große Lexika und Wörterbücher Europas. Europäische Enzyklopädien und Wörterbücher in historischen Porträts (2012) 193–210.
- [30] J. Ou, M. Vechev, O. Hilliges, An interactive system for data structure development, in: Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, CHI ’15, Association for Computing Machinery, 2015, pp. 3053–3062. doi:10.1145/2702123.2702319.
- [31] F. Beck, B. Dit, J. Velasco-Madden, D. Weiskopf, D. Poshyvanyk, Rethinking user interfaces for feature location, in: 2015 IEEE 23rd International Conference on Program Comprehension, IEEE, 2015, pp. 151–162. doi:10.1109/ICPC.2015.24.
- [32] M. Almeida, G. Cole, K. Du, G. Luo, S. Pan, Y. Pan, K. Qiu, V. Reddy, H. Zhang, Y. Zhu, C. Omar, RustViz: Interactively visualizing ownership and borrowing, in: 2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), IEEE, Roma, Italy, 2022, pp. 1–10. doi:10.1109/VL/HCC53370.2022.9833121.
- [33] S. Clerici, C. Zoltan, G. Prestigiacomo, Graphical and incremental type inference. a graph transformation approach, High.-order Symb. Comput. 26 (1-4) (2013) 29–62. doi:10.1007/s10990-014-9104-8.
- [34] M. Noone, A. Mooney, Visual and textual programming languages: a systematic review of the literature, Journal of Computers in Education 5 (2) (2018) 149–174. doi:10.1007/s40692-018-0101-5.
- [35] R. da Silva Ribeiro, L. de Oliveira Brandão, T. V. M. Faria, A. A. F. Brandao, Programming web-course analysis: How to introduce computer programming?, in: 2014 IEEE Frontiers in Education Conference (FIE) Proceedings, IEEE, 2014, pp. 1–8. doi:10.1109/FIE.2014.7044140.
- [36] D. C. Cliburn, Student opinions of alice in CS1, in: 2008 38th Annual Frontiers in Education Conference, IEEE, 2008, pp. T3B–1–T3B–6. doi:10.1109/FIE.2008.4720254.
- [37] T. Daly, Minimizing to maximize: An initial attempt at teaching introductory programming using alice, Journal of Computing Sciences in Colleges 26 (5) (2011) 23–30.
- [38] S. Montero, P. Díaz, D. Díez, I. Aedo, Dual instructional support materials for introductory object-oriented programming: Classes vs. objects, in: IEEE EDUCON 2010 Conference, IEEE, 2010, pp. 1929–1934. doi:10.1109/EDUCON.2010.5492438.
- [39] B. Alshaigy, S. Kamal, F. Mitchell, C. Martin, A. Aldea, PILeT: an interactive learning tool to teach python, in: Proceedings of the Workshop in Primary and Secondary Computing Education, WiPSCE ’15, Association for Computing Machinery, 2015, pp. 76–79. doi:10.1145/2818314.2818319.

Appendix A. User study tasks

We provide all the level settings we used in our user study. The tasks are open source and available for evaluation². However, this can be attempted locally with a Haskell interpreter

²<https://zerotohero.fly.dev>

or even with a pen and paper. The target type is the desired type signature for the function `zeroToHero`. The available functions show a list of functions that are allowed to be used in the implementation. It is not required to use all the available functions, and it is not forbidden to use any other functions or variables outside the provided functions; even the Haskell prelude is not available.

Appendix A.1. Level 1: Trial run

Target Type.

- `zeroToHero :: Zero a -> Hero a`

Available Functions.

- `f :: Zero a -> Hero a`

Possible Solution.

- `zeroToHero z = f z`

Appendix A.2. Level 2: Assembly required

Target Type.

- `zeroToHero :: Zero a -> Hero a`

Available Functions.

- `runZero :: Zero a -> a`
- `mkHero :: a -> Hero a`
- `(\$) :: (a -> b) -> a -> b`

Possible Solution.

- `zeroToHero z = mkHero (runZero z)`

Appendix A.3. Level 3: Which path?

Target Type.

- `zeroToHero :: Zero a -> Hero (a, a)`

Available Functions.

- `f1 :: Zero a -> Hero a`
- `f2 :: Zero a -> (a, a)`
- `f3 :: Hero a -> Hero (a, a)`
- `(\$) :: (a -> b) -> a -> b`
- `(.) :: (b -> c) -> (a -> b) -> a -> c`

Possible Solution.

- `zeroToHero z = f3 . f1 $ z`

Appendix A.4. Level 4: A repeating pattern

Target Type .

- `zeroToHero :: Zero a b -> Hero b b`

Available Functions.

- `f1 :: Zero a b -> Hero b a`
- `f2 :: Zero a a -> Hero a a`
- `f3 :: Zero a b -> Zero b a`
- `f4 :: Zero a b -> Zero b b`
- `($) :: (a -> b) -> a -> b`
- `(.) :: (b -> c) -> (a -> b) -> a -> c`

Possible Solution.

- `zeroToHero z = f2 . f4 $ z`

Appendix A.5. Level 5: A perfect pair

Target Type .

- `zeroToHero :: Zero a b -> Hero b b`

Available Functions.

- `fst :: (a, b) -> a`
- `snd :: (a, b) -> b`
- `f1 :: Zero a b -> Hero b a`
- `f2 :: Zero a a -> Hero a a`
- `f3 :: Zero a b -> Zero b a`
- `f4 :: Zero a b -> Zero b b`
- `($) :: (a -> b) -> a -> b`
- `(.) :: (b -> c) -> (a -> b) -> a -> c`

Possible Solution.

- `zeroToHero z = snd .f3 . f1 $ z`

Appendix A.6. Level 6: Monty Hall

Target Type .

- zeroToHero :: Zero a b c -> Hero c a

Available Functions.

- f1 :: Zero a b c -> Zero c b a
- f2 :: Zero a b c -> Zero a c c
- f3 :: Zero a b c -> Hero b c
- (\$) :: (a -> b) -> a -> b
- (.) :: (b -> c) -> (a -> b) -> a -> c

Possible Solution.

- zeroToHero z = f3 . f1 . f2 \$ z

Appendix A.7. Level 7: TIE fighter

Target Type .

- zeroToHero :: Zero a b c -> Hero c

Available Functions.

- f1 :: Zero a b c -> Hero (a -> b)
- f2 :: Zero a b c -> Hero (b -> c)
- f3 :: Zero a b c -> Hero a
- (<\$>) :: (a -> b) -> Hero a -> Hero b
- (<*>) :: Hero (a -> c) -> Hero a -> Hero c
- (\$) :: (a -> b) -> a -> b
- (.) :: (b -> c) -> (a -> b) -> a -> c

Possible Solution.

- zeroToHero z = f2 z <*> (f1 z <*> f3 z)

Appendix A.8. Level 8: The middle man

Target Type .

- zeroToHero :: (a -> d) -> (b -> d) -> (c -> d) -> Zero a b c -> Hero a d c

Available Functions.

- `f1 :: Zero a b c -> Zero c a b`
- `f2 :: Zero a b c -> Hero a b c`
- `fmap :: (c -> d) -> Zero a b c -> Zero a b d`
- `(\$) :: (a -> b) -> a -> b`
- `(.) :: (b -> c) -> (a -> b) -> a -> c`

Possible Solution.

- `zeroToHero ad bd cd z = f2 . f1 . f1 . fmap bd . f1 \$ z`

Appendix A.9. Level 9: Split the difference

Target Type .

- `zeroToHero :: Zero a b c d -> Hero d d d d`

Available Functions.

- `f1 :: Zero a b c d -> Zero a a b b`
- `f2 :: Zero a b c d -> Hero c c d d`
- `f3 :: Zero a b c d -> Zero d c b a`
- `(\$) :: (a -> b) -> a -> b`
- `(.) :: (b -> c) -> (a -> b) -> a -> c`

Possible Solution.

- `zeroToHero ad bd cd z = f2 \$ f1 \$ f1 \$ f3 \$ z`

Appendix A.10. Level 10: The roller coaster

Target Type .

- `zeroToHero :: Zero (a -> b -> c -> d) a b c -> Hero d`

Available Functions.

- `f1 :: Zero (a -> b) a c d -> Zero () b c d`
- `f2 :: Zero a b c d -> Zero b c d a`
- `f3 :: Zero a b c d -> Hero d`
- `(\$) :: (a -> b) -> a -> b`
- `(.) :: (b -> c) -> (a -> b) -> a -> c`

Possible Solution.

- `zeroToHero z = f3 . f2 . f2 . f1 . f2 . f1 . f2 . f1 \$ z`