

Just TYPEical: Visualizing Common Function Type Signatures in R

Cameron Moy * Julia Belyakova  Alexi Turcotte  Sara Di Bartolomeo  Cody Dunne 

Northeastern University

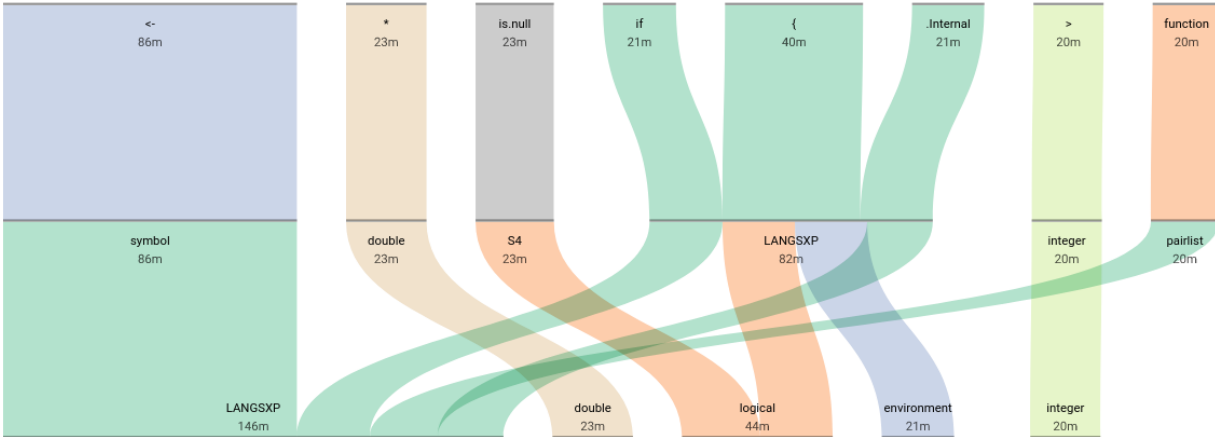


Figure 1: Our type flow visualization showing type signatures for a subset of R’s base package functions. Function names are listed at the top followed by the first two argument types. Complete signatures are shown in the full visualization (Fig. 2).

ABSTRACT

Data-driven approaches to programming language design are uncommon. Despite the availability of large code repositories, distilling semantically-rich information from programs remains difficult. Important dimensions, like run-time type data, are inscrutable without the appropriate tools. We contribute a task abstraction and interactive visualization, TYPEICAL, for programming language designers who are exploring and analyzing type information from execution traces. Our approach aids user understanding of function type signatures across many executions. Insights derived from our visualization are aimed at informing language design decisions — specifically of a new gradual type system being developed for the R programming language. A copy of this paper, along with all the supplemental material, is available at osf.io/mc6zt

Index Terms: Human-centered computing—Visualization

1 INTRODUCTION

Programming languages commonly evolve by decree. Often, the language designer decides that a new feature is necessary, or that a past feature was ill-conceived. Thus, the language moves forward — forcing its users to adapt to the changes. However, rarely is language design informed by empirical data on how programmers *actually write* software in practice [6].

Thanks to the prevalence of open source code, it is feasible to collect data on the use of popular programming languages. Vast quantities of code are publicly available on language-specific package servers. To inform programming language design, this collected data needs to be analyzed and interpreted. Programs are complex and highly structured, so researchers often employ static and dynamic

*E-mails: [[camoy](mailto:camoy@ccs.nyu.edu) | [belyakovay](mailto:belyakovay@ccs.nyu.edu) | [alex_i](mailto:alex_i@ccs.nyu.edu)]@ccs.nyu.edu, [[@northeastern.edu](mailto:dibartolomeo.s@c.dunne)]

analyses to gather information about specific aspects of programs. Even then, it may be difficult to make sense of the results of these analyses, especially if the data set is large.

Programming language design, and type system development in particular, can make use of run-time type signature information. A *type signature* describes the argument and return types a particular function is called with at run time. A *type system* provides a conservative approximation of run-time types. Understanding the frequency of type signatures in the wild is key for the development of new gradual type systems, whose adoption depends on integrating well with existing code. Without data-driven tools, type system designers are left to guess how their language is used in practice.

Our aim is to eliminate such guesswork by assisting designers during multiple phases of development. For example, exploratory analysis can identify unexpected edge cases or weed out language designs incompatible with existing code. We followed the Design Study “Lite” methodology [14] over 7 months to help the developers of a new gradual type system for the R programming language.

The contributions of this ongoing design study are:

- A *task abstraction* for programming language designers analyzing run-time type signatures for type system development.
- *The design and implementation of TYPEICAL*, an interactive visualization of run-time type signatures that supports: filtering data down to interpretable subsets; understanding argument and return types; and comparing type signatures.
- *Initial validation* of our system design with a usability study.

TYPEICAL builds on a data set of run-time type information recorded during the execution of test and example code from the most widely used libraries in the R ecosystem. Our visual design links two well-established visualizations, parallel sets [7] and Treemaps [4] [11], to view and navigate these type traces. While our design study focuses on R, TYPEICAL should be useful for analyzing any language where similar data are available.

A copy of this paper, source code, and data are available at osf.io/mc6zt, and a demo is online at typeical.github.io

Domain Goals	Search Task	Query Task	Consume Task	Abstract Task Description
<i>Find function</i>	<i>Locate</i>	<i>Identify</i>	<i>Discover</i>	<i>Finding a function</i> is a <i>locate</i> task, where the target function is known, but its location within the package hierarchy is not. Once the desired function is found, a user must be able to <i>identify</i> relevant data of interest such as call frequency.
<i>Determine types</i>	<i>Browse</i>	<i>Identify</i>	<i>Discover</i>	<i>Determining types</i> is a <i>browse</i> task, where the target type signature is unknown, but the location of the function of interest is known. A user must be able to <i>identify</i> type signature frequencies. For example, to infer if the function is polymorphic (accepts arguments of multiple different types) or a predicate (returns a single logical value).
<i>Compare signatures</i>	<i>Lookup</i>	<i>Compare</i>	<i>Discover</i>	<i>Comparing type signatures</i> is a <i>lookup</i> task where both target type signatures have already been located. A user must be able to discern how often a signature is used <i>compared</i> to another, both within a function and across different functions.

Table 1: Domain goals for programming language designers trying to understand function type signatures at run time. We captured these goals from interviews and created task abstractions based on Brehmer and Munzner’s multi-level task typology [3].

2 BACKGROUND AND PRIOR WORK

R is a lazy, multi-paradigm, dynamically-typed programming language widely used for statistical computing [9]. Unlike statically-typed languages, R code contains no type annotations and never passes through a typechecker, so many type-incorrect programs can be executed. For example, a time-consuming computation may accidentally contain a call $s + 1$ where s is a string. This will cause a run-time error “non-numeric argument to binary operator,” and the computation will halt — wasting time.

A *static type system* restricts arguments to the function $+$, detecting such errors before a program can begin execution. The goal is to design a *practical* static type system; one that allows most correct programs to compile, but is able to catch all type-incorrect programs ahead of time. A *gradual type system* permits integrating static and dynamic typing [12] [16], and is capable of seamlessly mixing typed and untyped regions of code. As a dynamically-typed language in widespread use, R is a good target for a gradual type system.

In statically-typed languages, the type system is usually an inseparable part of the language. Since they are designed at the same time, the type system precedes any code written in the language. Gradual type systems, by contrast, are often designed atop an existing dynamically-typed language [17]. Many R programs already exist, so a successful gradual type system must work well with extant code. Thus, gradual type system designers must first understand existing language idioms, and then work to accommodate them. Our tool TYPEICAL assists during the early phases of development, where a deep understanding of existing language use is necessary.

Code visualization, targeted at programmers attempting to understand a particular piece of software, has been widely studied. However, we are not aware of any previous work that applies visualization techniques for the purpose of programming language design. A language designer is concerned with understanding broad trends across the whole language ecosystem and not one specific program. Existing work, which narrows its analysis to a particular program, cannot yield the generalizable insights needed for type system development. Consequently, our system uses an entirely different set of visualization idioms compared to prior work.

Although the focus of prior work is different, their data are also derived from run-time execution information. For example, massive program execution traces are visualized in Bohnet et al. [1] to improve developer comprehension of large software systems, with pruning and summarization techniques being applied to reduce trace size. They focus on call topology, describing the structure of func-

tion invocations, rather than types. Telea et al. [15] also develop a methodology for displaying call structure. However, they visualize call graphs of systems in their entirety, rather than individual execution traces. TYPEICAL has elements of both, investigating the system as a whole, but doing so via execution traces. Another approach to call structure visualization is taken by Xie et al. [20], who employ dynamic call stack information rather than static call graphs. Besides node-link diagrams showing call graphs, they display execution times and the mapping of call stack trees to 2D space; this information is used for anomaly detection. In addition to standard call graphs, LaToza and Myers [8] adorn nodes and edges with extra information, allowing the user to gain further understanding of the system under examination. This includes grouping methods that come from the same class — a basic form of type data.

3 DATA AND COLLECTION

We used data from Turcotte et al. [18] that consists of traces from over 760,000 lines of R code and 534,000 lines of native code. This code was obtained from CRAN, a curated repository of R packages that is widely used in the community. The data set consists of all packages with at least 65% code coverage and at least 5 reverse dependencies (clients using that package). Out of the over 15,000 packages distributed on CRAN, this amounted to 400 packages.

Type traces were recorded by running a dynamic analysis over the test, example, and vignette code of each package. A *vignette* is a form of documentation that weaves together prose and executable code. A *type trace* is a list of function type signatures containing, among other attributes, the function name and types for each argument and return value. For example, the expression $0 < 1$ would be recorded as the type signature `integer → integer → logical`, meaning that the function `<` took two integer values and returned a logical value (i.e., a boolean). Types are assigned by the dynamic analysis and refine their actual run-time types. The analysis, for example, assigns the scalar tag `double` to a singleton vector of double values as R does not have native support for scalars. Our data set is a table of reduced type traces.

TYPEICAL has been instantiated with R type trace data, but the architecture is not specific to R at all. Any dynamically-typed language, or statically-typed language with run-time type information, could be instrumented to generate suitable data. As long as the trace output follows our schema, TYPEICAL will be able to visualize the data without any modifications.

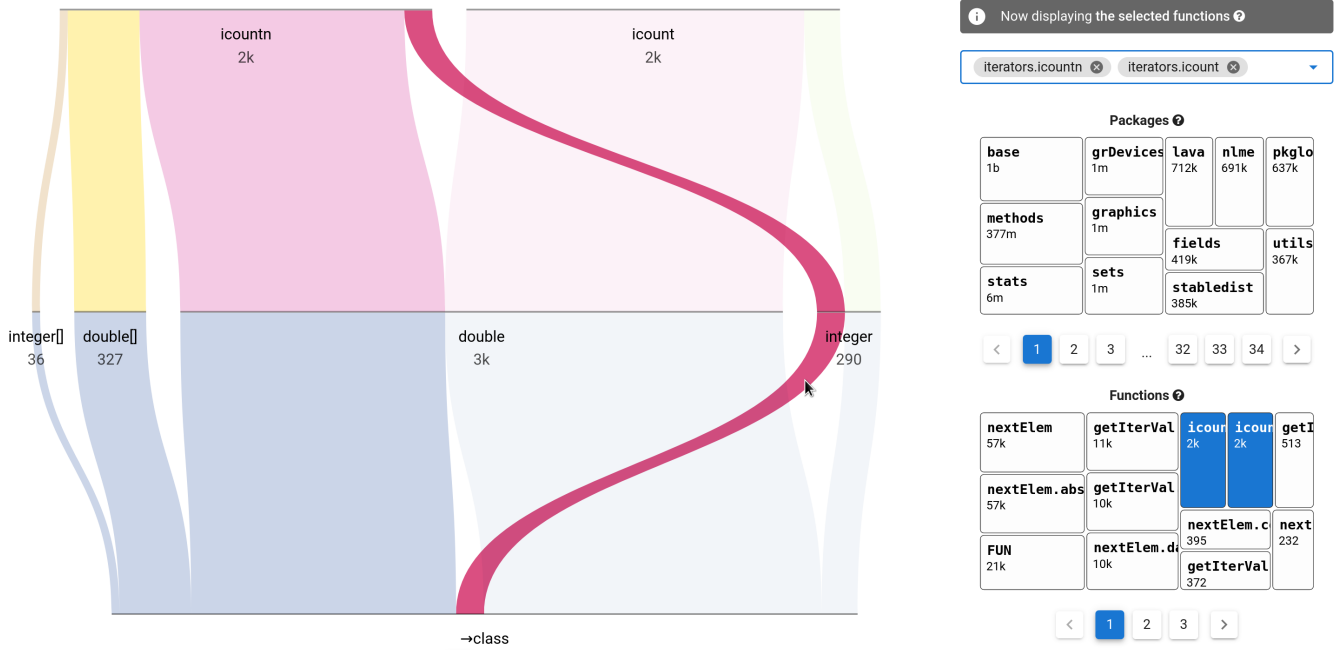


Figure 2: TYPEICAL, with the type flow visualization on the left and the filtering panel on the right. Here, the `icountn` and `icount` functions from the `iterators` package are both selected and displayed simultaneously to support *Compare* tasks. We see that the `icount` function is called with two different type signatures at run time. The function is usually used with the signature `double → class`. However, a substantial fraction of the time it is instead used as `integer → class`.

4 TASK ABSTRACTION

We interviewed the lead developer of a gradual type system for R (and co-author of this paper) to inventory relevant domain-specific tasks. Additionally, we sought his and other researchers’ feedback throughout the creation of TYPEICAL to ensure it adequately met requirements. These tasks are detailed and abstracted in Table 1.

From our interview, we learned of a common workflow for a type system designer: based on folklore knowledge about the language, they hypothesize that a widely-used function can be assigned a static type signature. To validate this hypothesis, they need to make sure that this type signature is compatible with uses found in the wild. They (1) perform the “find function” task and pull information about the function, (2) “determine types” of the function to learn the run-time type signatures, and (3) either validate or refute the original hypothesis. See Section 6.1 for a specific real-world example.

5 DESIGN AND IMPLEMENTATION

Using our abstracted tasks, we designed and developed TYPEICAL: an interactive web-based visualization of run-time type signatures.

Fig. 2 shows the heart of TYPEICAL — the type flow panel on the left and the filtering panel on the right. The system uses multiple coordinated views [19] [10], where signatures shown in the type flow can be interactively filtered to any selected subset of packages and functions. By default, the type flow panel shows several of the most frequent function type signatures in the entire data set. Our R type trace data, described in Section 3, contains over a million rows. Aggregation and filtering ensures TYPEICAL remains responsive.

5.1 Types as Flows

Our main visualization (Fig. 2, left) is a parallel sets [7] encoding that shows the type signatures of selected functions as flows. Each datum can be thought of as a tuple $(f, \tau_1, \dots, \tau_n, \tau_r)$ corresponding to the function name, argument types, and return type. Each tuple

component is categorical, and we are interested in visualizing each signature’s frequency. This is the exact use case for parallel sets.

Flows begin at a node labeling a function, curve across nodes for each argument type in order, and terminate at the return type. A width encoding shows how many times a function is called with that signature in the analysis (*Identify* tasks). Each flow segment is filled with a hue-varying categorical color scale, determined by the return type where the segment ends. To support the *Identify* tasks, each node label includes a count of either how many times the function was invoked (first row) or how often a value of that type was recorded during analysis (later rows).

Interactivity significantly enhances the usability of parallel sets. Hovering over a flow highlights its path and fades out all other functions, as shown in Fig. 2. When displaying many flows simultaneously, this highlighting becomes critical for user comprehension. Additional quantitative information about the flow is supplied on-demand when the user hovers over it. Clicking a flow will focus on that function and filter away all others; double-clicking outside all flows will bring back the previous view. The number of flows shown is limited to promote legibility, and pagination is used beyond a threshold; this option is configurable in the SETTINGS tab.

Proper flow layout is a necessary consideration for a legible visualization [13]. Without additional processing, flow overplotting can make the visualization impossible to understand. Edge crossing minimization algorithms have been well-studied; we employ two existing layout methods to maintain real-time performance. First, we attempt computing the globally optimal solution, minimizing the amount of flow crossing by solving a mixed-integer linear program [5]. Even for moderately sized graphs this can take too long to compute. Therefore, the optimal algorithm is terminated after 1 second if it has not finished, and the final layout is computed by minimizing local crossing between layers. We argue that it is better to have a reasonable layout quickly, rather than incur significant delay waiting for a perfect layout.

5.2 Hierarchical Package Visualization

The filtering panel of TYPEICAL (Fig. 2, right) contains three elements: a search bar and two flat Treemaps, one for packages and one for functions. These components support both the *Locate* and *Browse* search tasks in Table 1 for functions and packages. The search bar uses autocomplete and allows one to *Locate* a function among the thousands available. Alternatively, the Treemap views are more effective to *Browse* all available packages and functions.

Treemaps are often used for displaying hierarchical data [11], but in TYPEICAL they have a flat structure as packages in our data are not deeply nested. Many programming languages, like Java, contain deep namespace hierarchies in their package ecosystems. A more hierarchical Treemap would be appropriate in such cases, and TYPEICAL could be easily modified accordingly.

The Treemap area encodings convey the log-scaled frequency of calls to the associated function (Function Treemap) or all functions defined in that package (Package Treemap). Log-scaling, combined with pagination, keep nodes of the Treemaps and their labels legible, even when absolute frequency differences are significant. This is appropriate given our focus on supporting *Browse* tasks. We use the Squarified Treemap algorithm by Bruls et al. [4] to avoid rectangles with extreme aspect ratios that would make area perception less accurate and are harder to label.

All three filtering components are linked. Selection in the Packages Treemap filters the Functions Treemap to only functions defined in those packages; additionally, the type flow visualization updates to show the most frequent function type signatures for those packages. Likewise, selection in the Functions Treemap or search bar filters the type flow visualization to only those functions' signatures.


By default, TYPEICAL allows only one function or package to be selected at a time. An option in the SETTINGS tab allows multiple function selection. In the example from Fig. 2, the `icountn` and `icount` functions from the `iterators` package are both selected. These functions are rendered simultaneously in the type flow visualization, allowing the user to *Compare* them.

5.3 Technical Details

Since the amount of data a user may query is large (see Section 3), TYPEICAL is split into frontend and backend components. Data is stored in a SQLite database that has been indexed to achieve lookups as fast as possible. A Node.js server provides database access via a REST API to the client-side interactive visualization code. The client is written using D3 [2] and Vue, and is responsive to a variety of browser dimensions. This architecture avoids excessive memory consumption and speeds up load and query times by only transferring the required subset of data. Edge crossing minimization is calculated using web workers to allow early termination and avoid blocking the main browser thread. While we have chosen a specific implementation strategy, the design requirements do not mandate this software stack. Any visualization framework that supports Treemaps, parallel sets, and rich interactivity, should suffice.

6 EVALUATION

In accordance with the design study “lite” methodology [14], we conducted a small-scale qualitative usability study to validate our system design. Our $n = 18$ participants were Computer Science M.S. and Ph.D. students at our institution.

Many respondents had some trouble interpreting the visualization or at least commented that the learning curve was steep. Given that the domain is niche and the flow-based idiom for visualizing signatures is novel, this was not unexpected. To alleviate some of this confusion, we incorporated elements that give explicit explanations for different components. These appear as question mark icons  that, upon hovering, give a complete description of the corresponding component. For more details, we also provide an ABOUT page with step-by-step instructions and a demo video.

6.1 A Preliminary Case Study

The R type system designers are considering adding more sophisticated class and object types to their type system — hypothesizing that one umbrella type for all classes is inadequate. To get a better idea for this, they are interested in how classes flow through type traces. Specifically, how does the input class affect the output class?

Consider `dp1yr`'s `select` function that accepts an R data frame, a number of column names, and returns the specified columns. At first glance, the type is uninteresting, with a single type trace (`class` $\rightarrow \dots \rightarrow$ `class`) occurring about 70 times. With the “detailed data” setting enabled, however, the type becomes much more interesting; it reveals five different classes for the first argument, matching up perfectly with the five different classes for the return type. We see that `select` “preserves” the class of its first argument — passed an object of class `A`, it always returns an object of class `A`. This insight led the type system designers to conclude that richer class types would be useful to programmers.

7 DISCUSSION

Designed with a specific use case in mind, TYPEICAL suffers from some shortcomings and does not fully address the entire range of tasks a language designer may need to perform.

In particular, TYPEICAL has limited support for exploring type information at differing granularities. R has a range of data types with dimensionality information that we compress into a simpler type. Reducing the number of types can make the visualization more understandable, at the expense of losing precision in type information. At the moment, TYPEICAL supports interactive switching between fully compressed types and types with enhanced `class` and `list` information (e.g. `list<logical>`). Ideally, a user would be able to tune the level of granularity in multiple dimensions.

Filtering is a central mechanism of our system, but it also comes at a cost. If one has a specific function or several functions in mind, then TYPEICAL provides a useful local view of that information. It does not support, however, any kind of global view of type information across a large number of functions. When the amount of flows or nodes becomes too great, the visualization will start paginating, making comparisons across all flows substantially more difficult. Aggregation and summarization would be key to making a global view of the data feasible, but that remains future work.

8 CONCLUSION

We have presented a task abstraction and interactive visualization for analyzing and exploring run-time type information. Our tool, TYPEICAL, was instantiated with a massive data set of type traces from a corpus of popular R packages. The system is aimed at elucidating how R programmers use the language in the field. Specifically, the data and visualization will be used to inform the design of a future gradual type system for R. However, TYPEICAL is not tied to the R ecosystem and is suitable to use for any language that supports run-time type information. The visualization could yield insights useful for purposes beyond gradual typing and what we have considered.

We hope that future programming language designers will collect and use empirical data about their languages more extensively. Such data could be used to make informed decisions about design, where the legacy cost of making a mistake is high. Visualization will be unavoidable in these scenarios as semantically-meaningful information about programs is rich in structure. Generic, canned visualizations will not be sufficient. TYPEICAL is a first step in this direction.

ACKNOWLEDGMENTS

We thank Younes El Idrissi Yazami, Petr Maj, the students of CS 7250, and contributors to the open source projects we use. This work was funded in part by a Northeastern Graduate Fellowship and NSERC.

REFERENCES

- [1] J. Bohnet, M. Koeleman, and J. Doellner. Visualizing massively pruned execution traces to facilitate trace exploration. In *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pp. 57–64, Sep. 2009. doi: 10.1109/VISSOF.2009.5336416
- [2] M. Bostock, V. Ogievetsky, and J. Heer. D³: Data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011. doi: 10.1109/TVCG.2011.185
- [3] M. Brehmer and T. Munzner. A multi-level typology of abstract visualization tasks. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2376–2385, 2013. doi: 10.1109/TVCG.2013.124
- [4] M. Bruls, K. Huizing, and J. J. van Wijk. Squarified Treemaps. In *Data Visualization 2000*, pp. 33–42, 2000. doi: 10.1007/978-3-7091-6783-0_4
- [5] D. Ebner. Optimal crossing minimization using integer linear programming. Master’s thesis, Technische Universität Wien, 2005. <http://www.complang.tuwien.ac.at/cd/ebner/ebner05da.pdf>.
- [6] S. Hanenberg. Faith, hope, and love: An essay on software science’s neglect of human factors. In *Proc. ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA ’10*, p. 933–946, 2010. doi: 10.1145/1869459.1869536
- [7] R. Kosara, F. Bendix, and H. Hauser. Parallel Sets: interactive exploration and visual analysis of categorical data. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):558–568, 2006. doi: 10.1109/TVCG.2006.76
- [8] T. D. LaToza and B. A. Myers. Visualizing call graphs. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 117–124, 2011. doi: 10.1109/VLHCC.2011.6070388
- [9] F. Morandat, B. Hill, L. Osvald, and J. Vitek. Evaluating the design of the R language: Objects and functions for data analysis. In *European Conference on Object-Oriented Programming (ECOOP)*, 2012. doi: 10.1007/978-3-642-31057-7_6
- [10] C. North and B. Shneiderman. Snap-together visualization: A user interface for coordinating visualizations via relational schemata. In *Proc. Working Conference on Advanced Visual Interfaces, AVI ’00*, p. 128–135, 2000. doi: 10.1145/345513.345282
- [11] B. Shneiderman. Tree visualization with Tree-Maps: 2-d space-filling approach. *ACM Transactions on Graphics*, 11(1):92–99, 1992. doi: 10.1145/102377.115768
- [12] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, vol. 6, pp. 81–92, 2006.
- [13] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981. doi: 10.1109/TSMC.1981.4308636
- [14] U. H. Syeda, P. Murali, L. Roe, B. Berkey, and M. A. Borkin. Design study “lite” methodology: Expediting design studies and enabling the synergy of visualization pedagogy and social good. In *Proc. 2020 CHI Conference on Human Factors in Computing Systems, CHI ’20*, pp. 1–13, 2020. doi: 10.1145/3313831.3376829
- [15] A. Telea, H. Hoogendorp, O. Ersoy, and D. Reniers. Extraction and visualization of call dependencies for large C/C++ code bases: A comparative study. In *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pp. 81–88, 2009. doi: 10.1109/VISSOF.2009.5336419
- [16] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: from scripts to programs. In *OOPSLA ’06: Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, pp. 964–974, 2006. doi: 10.1145/1176617.1176755
- [17] S. Tobin-Hochstadt, M. Felleisen, R. B. Findler, M. Flatt, B. Greenman, A. M. Kent, V. St-Amour, T. S. Strickland, and A. Takikawa. Migratory typing: Ten years later. In *SNAPL*, pp. 17:1–17:17, 2017. doi: 10.4230/LIPIcs.SNAPL.2017.17
- [18] A. Turcotte, A. Goel, F. Křikava, and J. Vitek. Designing types for R, empirically. Under submission., 2020.
- [19] M. Q. Wang Baldonado, A. Woodruff, and A. Kuchinsky. Guidelines for using multiple views in information visualization. In *Proc. Working Conference on Advanced Visual Interfaces, AVI ’00*, p. 110–119, 2000. doi: 10.1145/345513.345271
- [20] C. Xie, W. Xu, and K. Mueller. A visual analytics framework for the detection of anomalous call stack trees in high performance computing applications. *IEEE Transactions on Visualization and Computer Graphics*, 25(1), 8 2018. doi: 10.1109/TVCG.2018.2865026