Designing a Sound Gradual Type System

Brendon Bown

WRTG 316, Section 3

May 31, 2023

Abstract

The type system of a language is an useful tool. A dynamic type system allows a programmer to iterate quickly, which is useful when creating a prototype. A static type system gives the programmer confidence that their program logic is sound, which is useful for larger, production-level codebases. However, the transition of the codebase from a dynamic type system to a static type system can be difficult, since it requires a full rewrite of the code in a new language. Gradual typing fills the gap between dynamic and static typing and allows the transition to be performed gradually. However, industry-level languages that are in use today, such as Dart and TypeScript, have *unsound* gradual type systems, which allows for error.

This paper provides a review of literature on the subject of *sound gradual typing*. First, it explores why Dart and TypeScript type systems are found to be unsound. Next, it describes how gradual type systems can be made sound using contracts at runtime. It then shows how contracts can be used in order to improve error handling in a gradually-typed language. After, it discusses how contracts can hurt performance and how the number of contracts used at runtime can be reduced through static analysis. Finally, it demonstrates that gradual typing can be extended beyond the simple models generally used in research, with the addition of classes, nullability checking, and type inference. Through this review, the paper comes to the conclusion that an application of the above concepts would allow one to create a gradual type system that is both sound and practical.

Table of Contents

Designing a Sound Gradual Type System

Introduction

One of the most important tools in a programmer's tool belt is often the programming language that they use. It is through that language that they are able to design solutions to the problems with which they are presented. Therefore the semantics and tooling of the language have a strong influence on how the solution is created.

Dynamic vs. Static Type Systems

A significant aspect of a language's semantics is its type system. Some languages have strict type systems. Every expression and value in the code must have a type assigned and the types must match what the language's compiler expects. If they don't match, the compiler won't allow the code to be compiled and run. This kind of type system is known as a *static type system*. The benefit of a static type system is that it allows code to be checked for correctness. This is especially important when the codebase is large and complex, since it is difficult for a programmer, or even a team of programmers, to check the codebase for correctness every time a change is made.

On the opposite end of the spectrum, there are languages that have lax type systems. In these systems, programmers don't have to specify the type of a value, so the type of a value isn't known by the computer until the program is running. This can cause the program to crash if a value isn't the type that the program expected it to be. However, this kind of type system, known as a *dynamic type system*, enables programmers to iterate quickly on their design without having to worry about proving the exact correctness of their program. Thus, these kinds of languages are useful for quickly creating prototypes and proofs of concept.

Both static and dynamic type systems are useful, and industry uses languages equipped with each. Dynamically-typed languages are great for starting an app and producing a minimum viable product. Statically-typed languages, on the other hand, work well for maintaining complex, interconnected systems. There are downsides to both, though. Dynamic languages can be difficult to debug and repair as the project gets bigger, since a lot of the logic has to be hand-checked by a programmer. Meanwhile, it can be difficult to start a project with a static language, since everything has to be specified from the beginning of the project, and changing that can be laborious. Thus, dynamic languages are good for the beginning of a project, while static languages are much better suited for later design.

Gradual Type Systems

It is inconvenient and costly, though, to start with a dynamic language, then rewrite everything in a static language once the project has grown big enough. This is where another kind of type system comes in handy: a *gradual type system*. In this kind of system, programmers are not required to specify types, in which case values are treated dynamically. This allows the fast iteration that comes with a dynamic language. However, the programmer *can* specify types wherever they choose, allowing them to use the compiler to check the types they specify. This provides the correctness of a static language. Thus, a gradual type system offers the best of both worlds.

Research Topic

While a gradual type system does a good job at bringing the good parts of dynamic and static programming languages together, it has some problems of its own. Specifically, a gradually-typed language has to deal both with code that is typed and code that is untyped, while still maintaining *soundness*. A *sound* type system is one that correctly accepts or rejects a program based on whether its type annotations are correct. Thus, a sound *gradual* type system has to ensure that typed code will run correctly even while untyped code cannot be checked for correctness. This can be a difficult task. For this reason, I have chosen to write a literature review about the following question: *how can one design a gradually-typed language that is both practical and sound?*

Organization

To begin, this review will outline the methodology that the review used in order to select literature. Next, it will discuss the literature that was found. It will lay out how gradual typing is used in practical programming languages and why those type systems are unsound. Then, it will discuss how types can be enforced by contracts in order to make a type system sound. Next, it will analyze how contracts can be used in order to improve error handling in a gradually-typed program. It will then explore how to ensure that gradual typing doesn't reduce performance. Next, it will discuss how gradual typing can be extended beyond the more simple theoretical basis that is generally researched. This will be demonstrated with classes and type inference. Finally, it will summarize the points and discuss areas of future work.

Methodology

The methodology of this literature review is described by the **database** that was searched, the **selection criteria** that was chosen, and the **organization** of the review itself.

Database

The database that was selected for this literature review is Scopus. The reason for selecting Scopus is two-fold. First, Scopus has a clean interface that is simple to use. It is easy to clearly specify the criteria that the documents must meet, and it allows for advanced searches that provided more specificity.

In addition, Scopus has a reputation for being a high-quality database. They have a rigorous vetting process that involves researchers from within the field of the content being vetted. They also have helpful metadata associated with each document.

Selection Criteria

When I began my search, I knew that I wanted to focus my research on type systems, so I started out by searching "type systems". This brought back over 13,000 results, which meant that this search was too general. In order to refine it, I focused only on literature in the subject area of computer science, which narrowed it down to around 6,500 documents.

Next, I decided to focus specifically on *gradual* type systems, rather than just any type system. As I researched a little more about gradual typing, I found that it was sometimes referred to as *optional typing*. Thus, I added in "gradual typ*" and "optional typ*" to the search query. This brought it down to 116 documents, which is much less, but still not quite manageable.

Finally, I noticed that the soundness of a gradual type system was often mentioned when discussing the boundaries between typed and untyped code in gradually-typed code. I therefore decided to add "sound" and "unsound" to the query. This resulted in 18 documents.

As a final filter, I looked at the abstract of each document, searching for descriptions of how the author investigated the boundaries in gradually-typed languages. After removing duplicates, this resulted in 16 articles that I could review.

The final query string that I used in Scopus was:

TITLE-ABS-KEY("type systems") AND (TITLE-ABS-KEY("gradual typ*") OR TITLE-ABS-KEY("optional typ*")) AND (TITLE-ABS-KEY("sound") OR TITLE-ABS-KEY("unsound")) AND (LIMIT-TO (SUBJAREA,"COMP"))

Organization

This literature review will approach the literature by theme. This is fitting because the different pieces of literature discussed different aspects of how soundness could be implemented in a gradually-typed language. As described in the introduction, the themes that this review will focus on are: **practical application and unsoundness**; **type enforcement and contracts**; **error handling and blame**; **performance**; and **gradual type system extensions**. Each theme will have a **Results** section, where the literature is described, and a **Discussion** section, where the ideas presented within the literature are discussed.

Practical Application and Unsoundness

While a lot of research for gradual typing is theoretical, it has also been adopted by industry in programming languages such as

Dart and TypeScript. Both of these languages have gradual type systems implemented, but they both have an interesting property: their type systems are *unsound*.

A type system is considered *sound* if it only allows programs whose types match up logically. Any types that would cause the program to enter into an invalid state are rejected. On the other hand, when a type system is *unsound*, it doesn't always correctly deduce the types. In other words, some programs may be declared as valid by the type checker, but they may result in a type error at runtime.

Results

Ideally, a type system should be sound. It is generally considered to be much more useful when a type system catches *all* type errors. However, interestingly enough, the type systems of Dart and TypeScript, two languages with gradual typing implemented, are both unsound.

TypeScript

In an effort to understand the type system of TypeScript more in depth, Gavin Bierman, Martin Abadi, and Mads Torgersen define a core semantics for TypeScript in their article *Understanding TypeScript*. In their development of the core semantics, they explore the sources of unsoundness within the language. However, they find that the unsoundness is not completely due to the fact that TypeScript is gradually-typed.

Instead, it comes from one of the goals of TypeScript, which is that there should be no way to tell if JavaScript code was created from TypeScript code. In other words, the TypeScript compiler shouldn't insert any generated code into the resulting JavaScript. This means that there can't be any runtime checks inserted at the boundary between typed and untyped code. If the TypeScript compiler can't verify that types are correct when it is compiling to JavaScript code, there may be type errors that go undetected until runtime. Thus, the unsoundness of the type system is jus as much a result of the goals of the language as it is the gradually-typed nature of the type system (Bierman, G., Abadi, M., & Torgersen, M., 2014).

Dart

Dart also has a type system that is gradually-typed and unsound. In fact, this unsoundness was actually intentional on the part of the developers. They wanted to create a type system that was intuitive to the developers, one that felt more usable.

In their article *Type Unsoundness in Practice: An Empirical Study of Dart*, Gianluca Mezzetti, Anders Moller, and Fabio Strocco explore the different sources of unsoundness, including function types and parameter types. They then perform experiments using different additions to the type system and measure the amount of type errors that the additions would cause in order to determine how often the "unsoundness" of the type system is being used in Dart programs.

The results of their experiment show that not all of the sources of unsoundness are being

used in a productive way and that some unsound choices could be replaced with sound choices without significantly increasing the type warnings and runtime errors (Mezzetti, G., Møller, A., & Strocco, F., 2016).

There has been a lot of work done to recommend improvements to the type system. In *Message Safety in Dart*, Erik Ernst, Anders Moller, Mathias Schwarz, and Fabio Strocco show how a stronger enforcement of types would reduce unsoundness in the language, creating what they describe as *message-safe programs* (Ernst, E., et al., 2017).

Thomas S. Heinze, Anders Moller, and Fabio Strocco add to this in *Type Safety Analysis for Dart*. They outline two uses of type annotations, filtering and specification, that allow a static analysis of the program to find type errors that wouldn't be caught by the standard Dart type checker. Using these can improve the type safety of Dart, and it can decrease the number of type checks required at runtime (Heinze, T. S., Møller, A., & Strocco, F., 2016).

Discussion

Although both of the previously-mentioned gradually-typed industry languages are unsound, the unsoundness doesn't seem to come directly from the fact that the type systems are gradually typed. Rather it seems to be a choice of the language developer in order to accomplish other goals of the language.

In TypeScript, the problem of unsoundness could be solved by adding checks at runtime. However, due to the goal of not wanting to add any generated code to the programmer's own code, they don't add these type checks.

In Dart, the developers wanted to make the type system more usable and intuitive, so they allowed some type casting that isn't logically sound.

In both cases, there are solutions to the problem of unsoundness. Adding in runtime checks and disallowing logically-unsound type checking would increase and potentially remove any unsoundness in the type system. Therefore, a gradual type system is not inherently unsound. It is instead the choice of language designers that make the language unsound.

Type Enforcement and Contracts

A sound type system is one in which the type of a value matches the annotation that it has been given by the programmer. This ensures that the code is always working with the data structures that it expects. Normally, in a statically-typed language, the verification of types occurs as the program is compiled. In a gradually-typed language, this isn't always possible, since sometimes the type of a value isn't known until runtime. Therefore, in gradual typing, some checking is often performed at runtime.

Results

In *Typed-Untyped Interactions: A Comparative Analysis*, Ben Greenman, Christos Dimoulas, and Matthias Felleisen lay out different possible methods of enforcing gradual typing at type boundaries at runtime:

- **Erasure**: no type checks are performed at runtime. The program checks what it can at compile time, then allows the program to fail if the types are incorrect at runtime.
- **Transient**: type checks only check to make sure that the *shape* of the value is correct. For example, if the type is a class *Foo* with a field *bar*, the type check will only make sure that the value has the field *bar*, but it won't verify if the value is of the class *Foo*.
- **Natural**: type checks focus on ensuring that types are valid throughout the entire program, rather than just in typed areas.
- **Concrete**: each value that is being passed around has a tag that describes the data (such as the class name or primitive type). Type checking checks this tag to make sure that the types match up correctly with the expected type (Greenman, B., Dimoulas, C., & Felleisen, M., 2023).

Other than the method of *erasure*, which eliminates type checking at runtime completely, each of these methods need a way to establish type checking at type boundaries. As Max S. New, Daniel R. Licata, and Amal Ahmed describe in *Gradual type theory*, one way to implement this is through *contract-based models*. In these models, each time a boundary between typed and untyped code appears, a *contract* is inserted. A contract is a check that ensures that the value that it is given has the correct structure. These ensure that the statically-typed portion of the code is handling data in the structure that it expects (New, M. S., Licata, D. R., & Ahmed, A., 2021).

However, contracts can be computationally expensive at runtime. Therefore, as Cameron Moy, Phuc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn reveal in *Corpse Reviver: Sound and Efficient Gradual Typing via Contract Verification*, not all contracts that are inserted are strictly necessary. Through a process of contract verification at compile time, it can be proven that some contracts will never be broken. Thus, those contracts can be removed from the runtime execution. This reduces the amount of work that is done at runtime (Moy, C., et al., 2021).

Discussion

One way to ensure that a gradual type system is sound is by implementing type enforcement by contracts. This allows the programmer to have confidence that the types that they are declaring are correct. Contracts allow for runtime verification that values match the expected type.

However, as noted previously, contracts can be expensive to verify at runtime and can slow down the execution of the code. Thus, it is important to reduce the number of contracts within a program. This can be done

through a static analysis of the inserted contracts in order to determine which contracts will never be broken. This makes contracts a more appealing option for implementing a sound gradual type system.

Error Handling and Blame

When working with both typed and untyped code, it isn't possible to check if types always match up correctly at compile time. Sometimes the untyped code will produce values that don't quite match the expected type. When this happens, the code must throw an error at runtime.

While the type mismatch may happen at a boundary, the detection of the error may happen later in the code. Thus, it is important to determine the source of the error in order to correctly inform the programmer of the error. This is done using *blame calculus.*

Results

In their article *Well-Typed Programs Can't Be Blamed*, Philip Wadler and Robert Findler first introduce the idea of the "blame calculus". Blame calculus is a mathematical language that takes the source code and adds casts at type boundaries. In their exploration, Wadler and Findler use the blame calculus, as well as a type system that they design, in order to prove that in all cases the blame for type errors lies with untyped code (Wadler, P., & Findler, R. B., 2009).

Building on the blame calculus, Wenjia Ye, Bruno C. d. S. Oliveira, and Xuejing Huang add implicit type conversions based on type annotations. Rather than having an intermediate language, type checking and inference is done based on the type annotations. This allows for a series of casts to be collapsed into two casts at most . This removes potential for type errors, in addition to decreasing the space requirements for type checks at runtime. However, this can also cause a loss of blame. To account for this, the authors create a blame recovery calculus that is slightly more liberal in what it allows than the previously mentioned blame calculus, but that allows one to take advantage of the decreased space requirements (Ye, W., Oliveira, B. C. D. S., & Huang, X., 2021).

Expanding on the idea of blame, Ben Greenman, Lukas Lazarek, Christos Dimoulas, and Matthias Felleisen show the practicality and value of blame by implementing it in Typed Racket. In their implementation, they use a "blame map", which tracks type casts and type checks as they occur. This provides a reference that the program can refer to if it encounters a type error. In their implementation though, it does not perform well, both in terms of space usage and time to complete (Greenman, B., et al., 2022).

Discussion

Blame calculus is a useful tool for making boundaries between typed and untyped code explicit. It shows where casts are happening, as well as where the cause of type errors will be based on those boundaries. And, as Wadler and Findler demonstrate, the cause of

type errors will always be in dynamic code. This conclusion makes sense because the types serve as a specification that needs to be met, while the dynamic code has no such specification. Treating blame in this way simplifies error handling, since the interpreter doesn't have to decide between to possible sources of error.

In addition to the improved error handling, Ye, Oliveira, and Huang provide semantics that remove the need for an intermediate language. This is significant because it decreases the amount of logical steps necessary to make a gradual type system sound, thus simplifying the blame process. It also has the added benefit of requiring less space.

However, while there do seem to be benefits in using blame to improve error handling, the efforts of researchers in the implementation of blame tracking. As shown by the research of Greeman, Lazarek, Dimonulas, and Felleisen, practical blame tracking can be expensive, both in time and space required. This is an area of research that needs more attention in order to improve the performance of error handling in gradual type systems.

Performance

While gradual typing may be an improvement for type systems conceptually, it is also important to consider more practical aspects, such as space and time required.

Results

A gradually-typed language will likely never pass a statically-typed language in this area. A complete knowledge of a program's types allows for optimizations that aren't otherwise possible.

A gradually-typed language, though, could be expected to have a better performance than a dynamically-typed language, or at least equal performance.

However, as David Herman, Aaron Tomb, and Cormac Flanagan point out in their article *Space-Efficient Gradual Typing*, gradual typing requires runtime checks in order to enforce sound gradual typing. These runtime checks can be expensive, both in time and space. This is especially true if there are a lot of boundaries between typed and untyped code. Thus, they develop a strategy for implementing runtime checks that reduces the amount of type checks required. This reduces the amount of space and time required to enforce type soundness. However, they develop only a theoretical foundation for the strategy and leave it up to future research to apply it to more advanced type systems (Herman, D., Tomb, A., & Flanagan, C., 2010).

In addition to the work of improving the performance, Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen lay out a system of testing performance in *How to Evaluate the Performance of Gradual Type Systems*. They design 20 different programs that test the

performance of a gradually-typed language in different ways, including in real-life applications. This allows them to analyze and find the performance bottlenecks in their code. In their article, they apply the benchmarks to Typed Racket. (Greenman, B. E. N., et al., 2019)

Discussion

Although the conceptual ideas of gradual typing are a benefit to the programmer, if the space and time performance is poor enough, the costs may outweigh the benefits. Thus, it is important to research how gradual typing can be implemented in a performance-conscious way. One good way to do this is by eliminating as many runtime-checks as possible.

It is also important to be testing the language for performance edge-cases in order to detect if there are any performance bottlenecks. For this purpose, one could use the performance benchmarks designed in *How to Evaluate the Performance of Gradual Type Systems*.

Gradual Type System Extensions

Although gradual typing is generally discussed with respect to simple types and simple functions, there is research that expands the scope of sound gradual typing.

Results

In *Sound Gradual Typing is Nominally Alive and Well*, Fabian Muehlboeck and Ross Tate discuss the plausibility of implementing nominal typing in gradual typing. Nominal typing is a kind of typing where values are typed according to a name given at compile time (as opposed to structural typing where values are typed by their structure). They demonstrate how type casts, where a value's type is changed from one type to another, can be used in order to facilitate interaction between typed and untyped code (Muehlboeck, F., & Tate, R., 2017).

As an addition to nominal typing, gradual typing can be extended to include object-oriented classes. Asumu Takikawa, Daniel Feltey, Earl Dean, Matthew Flatt, Robert Bruce Findler, Sam Tobin-Hochstadt, and Matthias Felleisen elaborate on this in their article *Towards Practical Gradual Typing*. They layout four principles that should guide this extension:

- Class types differ from object types
- Dynamic class composition requires structural types
- Object types need subtyping, class types need row polymorphism
- Soundness checks need opaque contracts and contract sealing

By following these principles, one can implement classes within a gradual type system (Takikawa, A., et al., 2015).

This can be taken even further with the implementation of first-class classes. A *first-class class* is a class whose definition itself can be treated as a value, in addition to the objects constructed by the class. In their work in *Gradual Typing for First-Class*

Classes, Asuma Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen develop a formal model of a type system with first-class classes. They then use the model in order to explore the type system and to prove its soundness (Takikawa, A., et al., 2012).

Beyond an implementation of gradually-typed object-oriented programming, other extensions have also been proven to be sound. In *Gradual Program Analysis for Null Pointers*, Sam Estep, Jenna Wise, Jonathan Aldrich, Eric Tanter, Johannes Bader, and Joshua Sunshine discuss how the problem of null pointers appear in the Java language. Then, using principles of gradual typing, they develop a static analysis tool that allows the compiler to statically verify programmer annotations of nullability (Estep, S., et al., 2021).

Finally, gradual typing has been shown to work with type inference. Type inference allows a compiler to figure out the types of a program by looking at the surrounding context. This simplifies the code and allows it to be more flexible. Jeremy G. Siek and Manish Vachharajani show how type inference can be implemented in *Gradual Typing with Unification-based Inference*. After explaining why previous failed attempts at type inference didn't work, they develop a gradual type system and implement an inference algorithm for it. They then prove that the inference algorithm leads to a sound type system (Siek, J. G., & Vachharajani, M., 2008).

Discussion

As has been shown through multiple areas of research, a gradual type system can be sound in more than just the simple cases. A type system can implement object-oriented concepts, such as classes and, more specifically, first-class classes, while still remaining sound. It can also verify nullability, and type inference can be implemented while remaining sound.

Therefore, the design of a gradually-typed language doesn't need to sacrifice soundness in order to add other features. There are many features that can improve the developer experience that can be implemented in a sound manner.

Future Work

Although this literature review was intended to cover only soundness in a gradual type system, many of the papers that mentioned soundness spoke in a more general way about the boundaries between typed and untyped code. Each of the topics that these papers explored, such as type enforcement and blame, could be explored more in depth.

In addition, after the papers on Dart's type system were published, Dart released a new version of the language in which they made steps towards improving the soundness of the type system. Research could be done to investigate the changes that were made, how much they improved the soundness of the type system, and how much they impacted the developer experience.

Conclusion

This review was written to analyze literature in order to answer the question: *how can one design a gradually-typed language that is both practical and sound?*

It began by analyzing current gradually-typed languages used in industry and identifying the sources of their unsoundness. These sources seemed to be due more to the goals and choices of the language designers than specifically to the nature of gradual typing itself.

Afterwards, it explored *contracts* as a way of enforcing soundness at runtime. It discussed different ways of implementing contracts, then analyzed how static analysis can be used to improve those contracts.

Next, it showed how to use contracts implement a concept known as *blame*. Blame allows the type system to track where the source of the type error originated. However, the implementation of blame that was given was shown to be lacking in performance. More research needs to be done in that area in order to improve the technique.

The next section discussed the performance of contracts in general. It showed how performance could be improved by eliminating certain contracts that were proved to be useless. In addition, a list of benchmarks were designed in order to measure the performance of a gradually-typed system.

In the final section, it explored extensions to the simple gradual type system that is generally explored in research. It looked at adding object-oriented features, such as classes. It also looked at application in nullability testing, as well as how type inference could be implemented in gradual typing.

Therefore, this review concludes that, based on the previous literature, it is possible to design a gradual type system that is both sound and practical. Through the use of contracts, one can enforce the soundness of the type system at runtime. In addition, it is possible to use static analysis in order to reduce the number of contracts required. And this all can be used to create not just a simple type system, but a type system equipped with tools such as classes and type inference.

References

Bierman, G., Abadi, M., & Torgersen, M. (2014). *Understanding TypeScript* https://doi.org/10.1007/978-3-662-44202-9_11 https://www.scopus.com/inward/record.uri?eid=2-s2.0-84905405258&doi=10.1007%2f978-3-662-44202-9_11&partnerID=40&md5=3556fc4216fd90bae8fe129de742e07c

Ernst, E., Møller, A., Schwarz, M., & Strocco, F. (2017). Message safety in dart. *Science of Computer Programming*, *133*, 51-73. https://doi.org/10.1016/j.scico.2016.06.006

Estep, S., Wise, J., Aldrich, J., Tanter, É, Bader, J., & Sunshine, J. (2021). Gradual program analysis for null pointers. Paper presented at the , *194* https://doi.org/10.4230/LIPIcs.ECOOP.2021.3 https://www.scopus.com/inward/record.uri?eid=2-s2.0-85115216555&doi=10.4230%2fLIPIcs.ECOOP.2021.3&partnerID=40&md5=b113673345111f254197e404b27052e5

Greenman, B. E. N., Takikawa, A. S. U. M. U., New, M. S., Feltey, D., Findler, R. B., Vitek, J. A. N., & Felleisen, M. (2019). How to evaluate the performance of gradual type systems. *Journal of Functional Programming*, https://doi.org/10.1017/S0956796818000217

Greenman, B., Dimoulas, C., & Felleisen, M. (2023). Typed-untyped interactions: A comparative analysis. *ACM Transactions on Programming Languages and Systems, 45*(1) https://doi.org/10.1145/3579833

Greenman, B., Lazarek, L., Dimoulas, C., & Felleisen, M. (2022). A transient semantics for typed racket. *Art, Science, and Engineering of Programming, 6*(2) https://doi.org/10.22152/programming-journal.org/2022/6/9

Heinze, T. S., Møller, A., & Strocco, F. (2016). Type safety analysis for dart. Paper presented at the *DLS 2016 - Proceedings of the 12th Symposium on Dynamic Languages, Co-Located with SPLASH 2016*, 1-12. https://doi.org/10.1145/2989225.2989226 https://www.scopus.com/inward/record.uri?eid=2-s2.0-85002157865&doi=10.1145%2f2989225.2989226&partnerID=40&md5=50b1f905e86290be1c502694a8af5bbf

Herman, D., Tomb, A., & Flanagan, C. (2010). Space-efficient gradual typing. *Higher-Order and Symbolic Computation*, *23*(2), 167.

Mezzetti, G., Møller, A., & Strocco, F. (2016). Type unsoundness in practice: An empirical study of dart. Paper presented at the *DLS 2016 - Proceedings of the 12th Symposium on Dynamic Languages, Co-Located with SPLASH 2016,* 13-24. https://doi.org/10.1145/2989225.2989227 https://www.scopus.com/inward/record.uri?eid=2-s2.0-85002487103&doi=10.1145%2f2989225.2989227&partnerID=40&md5=f3bc864e0d8df81c3b05115b41be573e

Moy, C., Nguyen, P. C., Tobin-Hochstadt, S., & Van Horn, D. (2021). Corpse reviver: Sound and efficient gradual typing via contract verification. *Proceedings of the ACM on Programming Languages, 5*(POPL) https://doi.org/10.1145/3434334

New, M. S., Licata, D. R., & Ahmed, A. (2021). Gradual type theory. *Journal of Functional Programming, 31* https://doi.org/10.1017/S0956796821000125

Muehlboeck, F., & Tate, R. (2017). Sound gradual typing is nominally alive and well. *Proceedings of the ACM on Programming Languages, 1*(OOPSLA) https://doi.org/10.1145/3133880

Siek, J. G., & Vachharajani, M. (2008). Gradual typing with unification-based inference. Paper presented at the *DLS'08: Proceedings of the 2008 Symposium on Dynamic Languages,* https://doi.org/10.1145/1408681.1408688 https://www.scopus.com/inward/record.uri?eid=2-s2.0-78650759970&doi=10.1145%2f1408681.1408688&partnerID=40&md5=cd02a891018d8044d00612b4a49893ff

Takikawa, A., Feltey, D., Dean, E., Flatt, M., Findler, R. B., Tobin-Hochstadt, S., & Felleisen, M. (2015). Towards practical gradual typing. Paper presented at the *Leibniz International Proceedings in Informatics, LIPIcs, , 37* 4-27. https://doi.org/10.4230/LIPIcs.ECOOP.2015.4 https://www.scopus.com/inward/record.uri?eid=2-s2.0-84958693232&doi=10.4230%2fLIPIcs.ECOOP.2015.4&partnerID=40&md5=c16d0e9b26172ddcdba41e149c781c1a

Takikawa, A., Strickland, T. S., Dimoulas, C., Tobin-Hochstadt, S., & Felleisen, M. (2012). Gradual typing for first-class classes ? Paper presented at the *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA,* 793-810. https://doi.org/10.1145/2384616.2384674 https://www.scopus.com/inward/record.uri?eid=2-s2.0-84869859201&doi=10.1145%2f2384616.2384674&partnerID=40&md5=18c22e0cf1a8dd13b0f8481f213b194e

Wadler, P., & Findler, R. B. (2009). *Well-typed programs can't be blamed* https://doi.org/10.1007/978-3-642-00590-9_1 https://www.scopus.com/inward/record.uri?eid=2-s2.0-67650189558&doi=10.1007%2f978-3-642-00590-9_1&partnerID=40&md5=7a01ba32a2f9b723afa454f047a68247

Ye, W., Oliveira, B. C. D. S., & Huang, X. (2021). Type-directed operational semantics for gradual typing. Paper presented at the *Leibniz International Proceedings in Informatics, LIPIcs, , 194* https://doi.org/10.4230/LIPIcs.ECOOP.2021.12 https://www.scopus.com/inward/record.uri?eid=2-s2.0-85115203711&doi=10.4230%2fLIPIcs.ECOOP.2021.12&partnerID=40&md5=ab2c8014e6c944f91e125c3a5509a1c7