From Optional to Gradual Typing via Transient Checks

Michael M. Vitousek Jeremy G. Siek
Indiana University
{mvitouse, jsiek}@indiana.edu

1. Introduction

Gradual typing [11, 16] has seen widespread use over the last several years, in both academic research [1, 2, 9, 10, 12–14] and industrial language design [5, 6, 8]. Many recent implementations of gradual typing are translations to an existing untyped target language. Some of these translation do not perform runtime checking, which is needed for gradual soundness, in part because of the complexity of implementing sound proxy-based casts in the target language of the translation. These languages, which we refer to as *optionally-typed* [4], instead perform static typechecking and then erase types for execution, without guarding the boundaries of statically-checked and dynamically typed regions of code. The transient approach to gradual typing [17], which inserts lightweight checks to ensure soundness, provides a straightforward way to implement sound gradual typing for this class of languages. In this abstract we report on our experience in adapting the transient approach, originally designed for Python, to implement sound gradual typing for TypeScript [8].

1.1 Optional Typing vs. Gradual Typing

Traditionally, gradual typing mixes static and dynamic typing within the same language, statically rejecting programs proven to be ill-typed at compile time and performing runtime checks to enforce boundaries between static and dynamic at runtime. However, with the development of languages like Hack [5], TypeScript [8], and Dart [6], a distinct approach to gradual typing has become apparent. These industrial languages provide a fully-featured static type system for static portions of the program, but do not perform any additional runtime checking to catch errors missed by the static analysis. This design results in a language that is unsound with respect to its static type system, but which avoids both the runtime performance overhead of checking and the challenge for language implementers of designing a sound system of runtime checks.

To distinguish this approach from the traditional approaches to gradual typing, which preserve soundness through runtime checks, we refer to this class of language as being *optionally typed*, following Bracha [4]. Optionally typed languages include Hack, TypeScript, Dart, Typed Clojure [3], and Typed Lua [7].

The reluctance of language designers to implement runtime checks in these languages is understandable, because of another common aspect of their design: all of these languages are implemented in terms of a translation to a target untyped language: JavaScript for TypeScript and Dart, PHP for Hack, Clojure for Typed Clojure, and Lua for Typed Lua. These languages are *spartan hosts*: they are more or less immutable from the perspective of the designer of the gradual language and they are limited in their support for proxies. Under the traditional guarded approach, proxies implement runtime checks from higher-order casts or casts on mutable values. As such, the guarded design is challenging to implement. In contrast, Racket is not a spartan host for Typed Racket — its impersonator and chaperone mechanisms allow the guarded approach to be implemented in a straightforward way.

1.2 The Transient Semantics

In the process of developing Reticulated Python, we developed the transient design for runtime checks as an alternative to the guarded approach [17]. Unlike guarded, transient does not perform any proxying or wrapping of runtime values. Instead, it pervasively inserts lightweight use-site checks throughout translated programs. These checks do not alter the checked values, but merely ensure that they shallowly conform to the type expected of them. This system is sound because checks are inserted at all elimination forms: for example, if a function f has static type $\mathsf{int} \to \mathsf{int}$, a check will be inserted at all of f's call sites to ensure that the result of the call is an int.

Although transient requires insertion of checks even in typed code, we found that the performance overhead of the transient design in Reticulated Python fell well within the $3/10$-usable standard suggested by Takikawa et al. [15] on our benchmarks. We also proved the soundness of the transient design (even in the presence of unmediated interaction with arbitrary foreign code) and developed blame tracking, giving transient the key features needed to be a viable design choice for gradual typing [18].

We designed the transient approach to sidestep difficulties in the guarded approach, proxies interfering with object equality in the presence of casts [17] and the difficulties in retaining soundness in the presence of foreign functions unequipped to interact with proxies [18]. However, our practical experiences implementing both transient and guarded in Reticulated Python showed something more: compared to guarded, transient is both very tractable to implement and is more amenable to Python idioms and language forms.

2. Transient vs. Guarded in Reticulated Python

Reticulated Python is a testbed for experimenting with different designs for gradual typing in Python [17].[1] This tool contained a static typechecker for Python programs with type annotations and a source-to-source translator that emits plain Python code. We used this to implement both transient and guarded for Python, and found transient to work well, while the guarded approach resulted in programs failing at runtime due to the use of proxies, which Python (as a spartan host) could not always treat transparently.

In addition to solving the difficulties raised by guarded proxies, the transient design is substantially easier to implement for Python than guarded. For example, in the guarded implementation, object proxies must check if accessed attributes are methods and, if so, unbind them from the underlying proxied object and rebind them to the proxy before returning them. In addition, because classes

[1] Reticulated Python is available at `https://github.com/mvitousek/reticulated`.

can be casted, guarded generates complex code paths that uses Python metaclasses to create class proxies. Furthermore, when objects are constructed from a proxy class, an empty object of the underlying class has to be constructed *without* invoking its constructor, before being passed into the `__init__` constructor of the proxy. Proxy classes, at least in Reticulated's implementation, cannot be inherited from, so the cast inserter must insert code to "unproxy" superclasses within class definitions and then reimpose the superclass' constraints on the new subclass.

In contrast, the implementation of transient contained in Reticulated Python is extremely straightforward, relying entirely on inspecting Python values and comparing them to type tags. At runtime, reflection is used to verify the presence of object attributes and determine the arity of functions, but classes and bound methods do not require special cases and the machinery of guarded's proxies is completely absent. The source-to-source translator inserts checks at the use sites of higher-order values (such as function calls and field lookups), but the overhead of implementing this type-directed check insertion was minimal compared to implementing the typechecker and code emitter in the first place.

3. From TypeScript to CheckScript

Our work on Reticulated Python indicated that, when using the transient design, implementing runtime checks is a very small percentage of the overall effort and codesize required to develop a gradually typed language in terms of a source-to-source translation. The actual static typechecker and the machinery used to translate the typed language into the untyped host language are far more complex and required a much greater implementation effort.

We also observed that TypeScript, as well as other optionally-typed languages, already has these challenging features implemented. TypeScript contains a fully-featured static type system and typechecker, and it emits JavaScript programs with types erased. It only lacks runtime checks when the JavaScript programs are executed. Therefore, we hypothesized that it would be straightforward to implement transient for TypeScript, by simply altering the JavaScript emitter to produce JavaScript code with transient runtime checks.

Despite being initially unfamiliar with the internals of the TypeScript compiler and only passingly familiar with TypeScript itself, we have made quick progress on implementing the transient approach for TypeScript, which we refer to as CheckScript.[2] CheckScript supports the full breadth of TypeScript's type system and AST nodes, and can insert checks into regular, unmodified TypeScript programs, which successfully detect runtime type errors. However, CheckScript is not entirely complete; it has been tested on many small examples and several small (but real) TypeScript programs. The main future work lies in correctly handling TypeScript's module system. CheckScript is syntactically identical to TypeScript, so we expect to be able to run regular TypeScript programs without needing to change type annotations.

TypeScript's source-to-source translator is not type-directed like that of Reticulated Python, so some additional work was required to link the typechecker and the JavaScript emitter to emit JavaScript code that performs the required runtime checks. However, the total number of lines of code that differ between TypeScript and CheckScript is less than 1,000—even though the TypeScript implementation as a whole consists of nearly 75,000 lines.

TypeScript's static type system is more mature and powerful than that of Reticulated Python, so we needed to extend the transient design to support all the types available in TypeScript. TypeScript, unlike Reticulated Python, provides polymorphic function types. We adapted the transient design to support type parameters by emitting an additional argument in the translated JavaScript function, which a typed caller will use to pass in a type tag to be checked against. If the function is called from dynamic code, this additional argument will default to the dynamic type tag (any). TypeScript also provides union and intersection types; implementing checks for these type forms was straightforward. We check the value against all the type tags in an intersection type and we check that the value matches at least one type tag in a union type.

4. Conclusions and Ongoing Work

Our work with CheckScript leads us to believe that the transient design is applicable to other optionally-typed languages as well, and going forward we aim to develop experimental transient implementations of Typed Clojure, Typed Lua, and Hack. In these languages, transient may provide a solution to the challenges that prevented the adoption of sound gradual typing. For example, the guarded approach may be impossible for sound variants of Typed Clojure due to implementation details of Clojure and the JVM. Guarded's implementation complexity is not limited to Python — most industry implementations of gradual typing like TypeScript and Dart do not guarantee soundness partially because of its implementation overhead compared to its perceived utility. The transient approach provides a lower-complexity design that still preserves soundness that may be more palatable in practical language design.

The designers of optionally-typed languages like TypeScript and Typed Clojure did not implement sound gradual typing for understandable reasons — under the traditional guarded design, the tradeoff between increased detection of bugs versus increased runtime and implementation overhead does not clearly favor implementing sound runtime checks. Transient may tip this scale in the direction of soundness, because of the straightforwardness of its implementation in an optionally-typed language. However, to better understand this tradeoff, we must also determine how much runtime soundness buys programmers — whether the errors it detects are few and far between, or are frequent and challenging. As a gradually typed language with a large existing codebase, CheckScript will help us answer this question, and we hope to use it to learn whether or not optional typing is truly the wisest approach to designing languages that mix static and dynamic typing.

[2] CheckScript is available at `https://github.com/mvitousek/checkscript`.

References

[1] A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. Blame for all. In *POPL '11: Symposium on Principles of programming languages*, POPL '11, pages 1–14, 2011.

[2] E. Allende, O. Callaú, J. Fabry, E. Tanter, and M. Denker. Gradual typing for smalltalk. *Science of Computer Programming*, August 2013.

[3] A. Bonnaire-Sergeant, R. Davies, and S. Tobin-Hochstadt. *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, chapter Practical Optional Types for Clojure, pages 68–94. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. ISBN 978-3-662-49498-1. . URL `http://dx.doi.org/10.1007/978-3-662-49498-1_4`.

[4] G. Bracha. Pluggable type systems. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

[5] Facebook. Hack, 2013. URL `http://hacklang.org`.

[6] Google. Dart: structured web apps, 2011. URL `http://dartlang.org`.

[7] A. M. Maidl, F. Mascarenhas, and R. Ierusalimschy. Typed Lua: An optional type system for Lua. In *Proceedings of the Workshop on Dynamic Languages and Applications*, Dyla'14, pages 3:1–3:10, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2916-3. . URL `http://doi.acm.org/10.1145/2617548.2617553`.

[8] Microsoft. Typescript, 2012. URL `http://www.typescriptlang.org/`.

[9] A. Rastogi, A. Chaudhuri, and B. Hosmer. The ins and outs of gradual type inference. In *Symposium on Principles of Programming Languages*, POPL, pages 481–494, January 2012.

[10] B. M. Ren, J. Toman, T. S. Strickland, and J. S. Foster. The ruby type checker. In *SAC'13 (OOPS)*, 2013.

[11] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, September 2006.

[12] J. G. Siek, M. M. Vitousek, M. Cimini, S. Tobin-Hochstadt, and R. Garcia. Monotonic references for efficient gradual typing. In *ESOP '15*, April 2015.

[13] N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P.-Y. Strub, and G. Bierman. Gradual typing embedded securely in javascript. In *ACM Conference on Principles of Programming Languages (POPL)*, January 2014.

[14] A. Takikawa, T. S. Strickland, C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Gradual typing for first-class classes. In *Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 793–810, 2012.

[15] A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek, and M. Felleisen. Is sound gradual typing dead? In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 456–468, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. . URL `http://doi.acm.org/10.1145/2837614.2837630`.

[16] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium*, 2006.

[17] M. M. Vitousek, A. M. Kent, J. G. Siek, and J. Baker. Design and evaluation of gradual typing for Python. In *Symposium on Dynamic Languages*, DLS '14, pages 1–16, 2014.

[18] M. M. Vitousek, C. Swords, and J. G. Siek. Big types in little runtime, under consideration for OOPSLA 2016, 2016.