Tackling the Efficiency Problem of Gradual Typing

Esteban Allende *
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile
eallende@dcc.uchile.cl

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors

General Terms Languages, Performance

Keywords gradual typing, casts, performance

1. Extended abstract

The popularity of dynamic languages and their use in the construction of large and complex software systems makes the possibility to fortify grown prototypes or scripts using the guarantees of a static type system appealing. While research in combining static and dynamic typing started more than twenty years ago, recent years have seen a lot of proposals of either static type systems for dynamic languages, or partial type systems that allow a combination of both approaches [2–6, 8, 10, 14].

Gradual typing [11, 12] is a partial typing technique proposed by Siek and Taha that allows developers to define which sections of code are statically typed and which are dynamically typed, at a very fine level of granularity, by selectively placing type annotations where desired. The type system ensures that dynamic code does not violate the assumptions made in statically-typed code. This makes it possible to choose between the flexibility provided by a dynamic type system, and the robustness of a static type system.

The semantics of a gradually-typed language is typically given by translation to an intermediate language with casts, *i.e.* runtime type checks that control the boundaries between typed and untyped code. A major challenge in the adoption of gradually-typed languages is the cost of these casts, especially in a higher-order setting. Theoretical approaches have been developed to tackle the space dimension [7, 13], but execution time is also an issue. This has led certain languages to favor a coarse-grained integration of typed and untyped code [15] or to consider a weaker form of integration that avoids costly casts [16].

* Esteban Allende is funded by a CONICYT-Chile Ph.D. Scholarship.

SPLASH '13, October 26–31, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-1995-9/13/10.
http://dx.doi.org/10.1145/2508075.2514884

Other approaches include the work of Rastogi *et al.*. [10], using local type inference to significantly reduce the number of casts that are required; the work of Herman *et al.* [7], in which they propose to use coercions instead of proxies in a chain of higher-order cast, so as to be able to combine adjacent coercions in order to limit space consumption; the work of Siek *et al.* [13], in which they go a step further, developing threesomes as a data structure and algorithm to represent and normalize coercions. A threesome is a cast with three positions: source, target, and an intermediate lowest type. Combining a sequence of threesomes is done by taking the greatest lower bound of the intermediate types.

In developing Gradualtalk[1], a gradually-typed Smalltalk, our first concern was the design of the gradual type system, with its various features [1]. However, in the current stage of this work, we are concerned with the efficiency of casts. There are two concerns about casts insertions that we like to tackle.

The first concern is where to insert casts, especially those relate to method invocations. This is because method invocations are naturally very frequent in object-oriented programs, especially in pure object-oriented languages like Smalltalk. In the foundational paper on gradually-typed objects [12], Siek and Taha describe the semantics of cast insertion using a caller-side strategy—which we term the *call strategy*. Due to implementation issues, our first implementation of cast insertion was however based on a different approach, which we name the *execution strategy*. Here, casts are inserted on the callee side, at the beginning of each typed method. Studying the performance of both approaches revealed that they have complementary strengths, and that a third approach, which we call the *hybrid strategy*, could combine the best of both approaches.

We evaluated all three strategies with both microbenchmarks and macrobenchmarks. The microbenchmarks were designed to test the best and worst case of the execution strategy and call strategy and see how the hybrid strategy perform in those cases. The objective of the macrobenchmarks is to see how well these strategies translate to real world applications. Both benchmark confirm that the hybrid strategy performs as good as its best competitor in all cases.

The second concern is controlling when inserting casts. A gradual type system make the assumption that all the values can be implicitly casted from or to a dyn type, inserting the necessary cast to ensure type safety. However, this could not be the desire of the programmer. The programmer could have forgotten to type a variable or incorrectly pass the value to an untyped method without is knowledge. For normal methods or values, the additional cost of cast is negligible. However, for critical methods who are called multiple times, or blocks value where a cast insert a wrapper to the

[1] `http://www.pleiad.cl/gradualtalk`

block, the additional cost in performance can be significant. In that case, the programmer could desire to disable the automatic cast between the statically typed and dynamically typed world and be warned at compile time. However, the programmer does not have that choice in the gradual typing described by Siek and Taha.

Following the philosophy of gradual typing, disabling the insertion of implicit cast should be done in a fine grained way. The deactivation can be done from both ways: disallowing the automatic casting from the dynamically typed world to the statically typed world, which is annotated with $!\mathsf{T}$, and viceversa, *i.e.* disallowing automatic casting from statically typed to dynamically typed, which is annotated with $\sim\mathsf{T}$.

Both of these operators, ! and $\sim$, have practical uses. The operator ! can be used in the parameter type of a method to enforce that the method could be called only with statically typed variables, catching at compile time all clients who are trying to use the method with a dynamically typed variable. The operator $\sim$ can be used in an instance or local variable type to catch at compile time if a value is being inserted incorrectly in the dynamically typed world.

To describe how these operators work formally, we need to introduce the relationships of consistency and consistency subtype. Gradual typing extends traditional subtyping to *consistent subtyping* [12]. Consistency, denoted $\sim$, is a relation that accounts for the presence of Dyn: Dyn is consistent with any other type and any type is consistent with itself. The consistency relation is not transitive in order to avoid collapsing the type relation [11]. A type σ is a consistent subtype of τ, noted $\sigma \lesssim \tau$, iff either $\sigma <: \sigma'$ and $\sigma' \sim \tau$ for some σ', or $\sigma \sim \sigma''$ and $\sigma'' <: \tau$ for some σ''. A cast is inserted when an operation requires that a value of type τ is consistent subtype of a σ type, but is not strictly subtype.

In the case of the two operators ! and $\sim$, the type $!\mathsf{T}$ is consistent with Dyn, but Dyn is not consistent with the type $!\mathsf{T}$. For the operator $\sim$ is in the inverse order: Dyn is consistent with $\sim\mathsf{T}$, but $\sim\mathsf{T}$ is not consistent with Dyn. Looking at the description of consistency, these two operators make two important changes to the consistency relationship: there are types that that are not consistent with Dyn and the consistency relation is now asymmetrical. The consistency rules for the ! operator makes that Dyn is not a consistent subtype of $!\mathsf{T}$, raising a type check error instead of inserting a cast when a Dyn typed value is assigned to a $!\mathsf{T}$ typed variable, but still allowing that a value of type $!\mathsf{T}$ can be assigned (and automatically casted) to a Dyn typed variable. For the $\sim$ operator is the same principle, but in inverse order.

Both of these two concerns are complementary. The restricted automatic casts permits to choose when automatic cast insertion is not desired maintaining the flexibility of gradual typing, while the hybrid cast insertion strategy increase the performance of the application when automatic checks are used. We still need to fully formalize the two operators ! and $\sim$, how they relate with the different kinds of types and what is the impact of making the consistency relationship asymmetrical. We believe that tacking these two concerns allows that gradual typing can be used in real world applications, where debugging a big application can be a daunting task, without the concern of significant sacrifice of the performance of those applications.

References

[1] E. Allende, O. Callaú, J. Fabry, É. Tanter, and M. Denker. Gradual typing for Smalltalk. *Science of Computer Programming*, 2013. To appear.

[2] B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strniša, J. Vitek, and T. Wrigstad. Thorn: robust, concurrent, extensible scripting on the JVM. In *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2009)*, pages 117–136, Orlando, Florida, USA, Oct. 2009. ACM Press.

[3] G. Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, pages 1–6, 2004.

[4] G. Bracha and D. Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings of the 8th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 95)*, pages 215–230, Washington, D.C., USA, Oct. 1993. ACM Press. ACM SIGPLAN Notices, 28(10).

[5] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, pages 278–292, Toronto, Ontario, Canada, 1991.

[6] M. Furr. *Combining Static and Dynamic Typing in Ruby*. PhD thesis, University of Maryland, 2009.

[7] D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. *Higher-Order and Sympolic Computation*, 23(2):167–189, June 2010.

[8] K. Knowles and C. Flanagan. Hybrid type checking. *ACM Transactions on Programming Languages and Systems*, 32(2):Article n.6, Jan. 2010.

[9] POPL 2010. *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2010)*, Madrid, Spain, Jan. 2010. ACM Press.

[10] A. Rastogi, A. Chaudhuri, and B. Hosmer. The ins and outs of gradual type inference. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2012)*, pages 481–494, Philadelphia, USA, Jan. 2012. ACM Press.

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

[12] J. Siek and W. Taha. Gradual typing for objects. In E. Ernst, editor, *Proceedings of the 21st European Conference on Object-oriented Programming (ECOOP 2007)*, number 4609 in Lecture Notes in Computer Science, pages 2–27, Berlin, Germany, july/august 2007. Springer-Verlag.

[13] J. Siek and P. Wadler. Threesomes, with and without blame. In POPL 2010 [9], pages 365–376.

[14] S. Tobin-Hochstadt. *Typed Scheme: From Scripts to Programs*. PhD thesis, Northeastern University, Jan. 2010.

[15] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008)*, pages 395–406, San Francisco, CA, USA, Jan. 2008. ACM Press.

[16] T. Wrigstad, F. Zappa Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In POPL 2010 [9], pages 377–388.