Abstracting Gradual References

Matías Toro
PLEIAD Laboratory, Computer Science Department (DCC), University of Chile, Santiago, Chile
mtoro@dcc.uchile.cl

Éric Tanter
PLEIAD Laboratory, Computer Science Department (DCC), University of Chile, Santiago, Chile
etanter@dcc.uchile.cl

Abstract

Gradual typing is an effective approach to integrate static and dynamic typing, which supports the smooth transition between both extremes via the (programmer-controlled) precision of type annotations [19, 21]. Imprecision is normally introduced via the unknown type ?, e.g. function type Int → Bool is more precise than ? → ?, and both more precise than ?. Gradual typing relates types of different precision using consistent type relations, such as *type consistency* (resp. *consistent subtyping*), the gradual counterpart of type equality (resp. subtyping). For instance, ? → Int is consistent with Bool → ?. This approach has been applied in a number of settings, such as objects [20], subtyping [20, 11], effects [4, 5], ownership [18], typestates [27, 12], information-flow typing [9, 10, 23], session types [14], refinements [17], set-theoretic types [6], Hoare logic [3], parametric polymorphism [1, 2, 16, 15, 28, 24], and references [19, 13, 22].

In particular, gradual typing for mutable references has seen the elaboration of various possible semantics: *invariant* references [19], *guarded* references [13], *monotonic* references [22], and *permissive* references [22]. Invariant references are a form of references where reference types are invariant with respect to type consistency. Guarded references admit variance thanks to systematic runtime checks on reference reads and writes; the runtime type of an allocated cell never changes during execution. Guarded references have been formulated in a space-efficient coercion calculus, which ensures that gradual programs do not accumulate unbounded pending checks during execution. Hereafter, we refer to this language as HCC. Monotonic references favor efficiency over flexibility by only allowing reference cells to vary towards more precise types. This allows reference operations in statically-typed regions to safely proceed without any runtime checks. Permissive references are the most flexible approach, in which reference cells can be initialized and updated to any value of any type at any time.

These four developments reflect different design decisions with respect to gradual references: is the reference type constructor variant under consistency? Can the programmer specify a precise bound on the static type of a reference, and hence on the corresponding heap cell type? Can the heap cell type evolve its precision at runtime, and if yes, how? There is obviously no absolute answer to these questions, as they reflect different tradeoffs such as in efficiency and precision. This work explores the semantics that results from the application of a *systematic* methodology to gradualize static type systems. Currently we can find in the literature two methodologies to gradualize statically-typed languages: Abstracting Gradual Typing (AGT) [11], and the Gradualizer [7]. In this work, we consider the AGT methodology as it naturally scales to auxiliary structures such as a mutable heap.

The AGT methodology helps to systematically construct gradually-typed languages by using abstract interpretation [8] at the type level. In brief, AGT interprets gradual types as an abstraction of sets of possible static types, formally captured through a Galois connection. The static semantics of a gradual language are then derived by lifting the semantics of a statically-typed language through this connection, and the dynamic semantics follow by Curry-Howard from proof normalization of the type safety argument. The AGT methodology has been shown to be effective in many contexts: records and subtyping [11], type-and-effects [4, 5], refinement types [17, 26], set-theoretic and union types [6, 25], information-flow typing [23], and parametric polymorphism [24]. However, this methodology has never been applied to mutable references in isolation. Although Toro *et al.* [23] apply AGT to a language with references, they only gradualize *security levels* of types (*e.g.* Ref $\mathsf{Int}_{?}$), not whole types (*e.g.* Ref ? is not supported). In this article we answer the following open questions:

34th European Conference on Object-Oriented Programming (ECOOP 2020).
Editors: Robert Hirschfeld and Tobias Pape; Article No. 33; pp. 33:1–33:4
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Which semantics for gradually-type references follows by systematically applying AGT? Does AGT justify one of the existing approaches, or does it suggest yet another design? Can we recover other semantics for gradual references, if yes, how?

This article first reviews the different existing gradual approaches to mutable references through examples. It then presents the semantics for gradual references that is obtained by applying AGT, and how to accommodate the other semantics. More specifically, this work makes the following contributions:

- We present $\lambda_{\widetilde{\mathrm{REF}}}$, a gradual language with support for mutable references. We derive $\lambda_{\widetilde{\mathrm{REF}}}$ by applying the AGT methodology to a fully-static simple language with mutable references called λ_{REF}. This is the first application of AGT that focuses on gradually-typed mutable references.
- We prove that $\lambda_{\widetilde{\mathrm{REF}}}$ satisfies the gradual guarantee of Siek *et al.* [21]. We also present the first formal statement and proof of the conservative extension of the dynamic semantics of the static language [21], for a gradual language derived using AGT.
- We prove that the derived language, $\lambda_{\widetilde{\mathrm{REF}}}$, corresponds to the semantics of guarded references from HCC. Formally, given a $\lambda_{\widetilde{\mathrm{REF}}}$ term and its compilation to HCC$^+$ (an adapted version of HCC extended with conditionals and binary operations) we prove that both terms are bisimilar, and that consequently they either both terminate, both fail, or both diverge.
- We observe that $\lambda_{\widetilde{\mathrm{REF}}}$ and HCC$^+$ differ in the order of combination of runtime checks. As a result, HCC is space efficient whereas $\lambda_{\widetilde{\mathrm{REF}}}$ is not: we can write programs in $\lambda_{\widetilde{\mathrm{REF}}}$ that may accumulate an unbounded number of checks. We formalize the changes needed in the dynamic semantics of $\lambda_{\widetilde{\mathrm{REF}}}$ to achieve space efficiency. This technique to recover space efficiency is in fact independent from mutable references, and is therefore applicable to other gradual languages derived with AGT.
- We formally describe how to support other gradual reference semantics in $\lambda_{\widetilde{\mathrm{REF}}}$ by presenting $\lambda^{\mathsf{pm}}_{\widetilde{\mathrm{REF}}}$, an extension that additionally supports both *permissive* and *monotonic* references. Finally, we prove for the first time that monotonic references satisfy the dynamic gradual guarantee, a non-trivial result that requires careful consideration of updates to the store.

Additionally, we implemented $\lambda_{\widetilde{\mathrm{REF}}}$ as an interactive prototype that displays both typing derivations and reduction traces. All the examples mentioned in this paper are readily available in the online prototype available at `https://pleiad.cl/grefs`.

As a result, this paper sheds further light on the design space of gradual languages with mutable references and contributes to deepening the understanding of the AGT methodology.

2012 ACM Subject Classification Theory of computation → Type structures; Theory of computation → Program semantics

Keywords and phrases Gradual Typing, Mutable References, Abstract interpretation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.33

Category SCICO Journal-first

Related Version A full version of the paper is available at `https://doi.org/10.1016/j.scico.2020.102496`.

Funding *Matías Toro*: Partially funded by FONDECYT Project 3200583.
Éric Tanter: Partially funded by FONDECYT Project 1190058.

References

1 Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*, pages 201–214, Austin, Texas, USA, January 2011. ACM Press.

2 Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. Theorems for free for free: Parametricity, with and without types. *Proceedings of the ACM on Programming Languages*, 1(ICFP):39:1–39:28, September 2017.

3 Johannes Bader, Jonathan Aldrich, and Éric Tanter. Gradual program verification. In Işil Dillig and Jens Palsberg, editors, *Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2018)*, volume 10747 of *Lecture Notes in Computer Science*, pages 25–46, Los Angeles, CA, USA, January 2018. Springer-Verlag.

4 Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. A theory of gradual effect systems. In *Proceedings of the 19th ACM SIGPLAN Conference on Functional Programming (ICFP 2014)*, pages 283–295, Gothenburg, Sweden, September 2014. ACM Press.

5 Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. Gradual type-and-effect systems. *Journal of Functional Programming*, 26:19:1–19:69, September 2016.

6 Giuseppe Castagna and Victor Lanvin. Gradual typing with union and intersection types. *Proceedings of the ACM on Programming Languages*, 1(ICFP):41:1–41:28, September 2017.

7 Matteo Cimini and Jeremy Siek. The gradualizer: a methodology and algorithm for generating gradual type systems. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, pages 443–455, St Petersburg, FL, USA, January 2016. ACM Press.

8 Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages (POPL 77)*, pages 238–252, Los Angeles, CA, USA, January 1977. ACM Press.

9 Tim Disney and Cormac Flanagan. Gradual information flow typing. In *International Workshop on Scripts to Programs*, 2011.

10 Luminous Fennell and Peter Thiemann. Gradual security typing with references. In *Proceedings of the 26th Computer Security Foundations Symposium (CSF)*, pages 224–239, June 2013.

11 Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, pages 429–442, St Petersburg, FL, USA, January 2016. ACM Press.

12 Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of typestate-oriented programming. *ACM Transactions on Programming Languages and Systems*, 36(4):12:1–12:44, October 2014.

13 David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. *Higher-Order and Sympolic Computation*, 23(2):167–189, June 2010.

14 Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, and Philip Wadler. Gradual session types. *Proceedings of the ACM on Programming Languages*, 1(ICFP):38:1–38:28, September 2017.

15 Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. On polymorphic gradual typing. *Proceedings of the ACM on Programming Languages*, 1(ICFP):40:1–40:29, September 2017.

16 Lintaro Ina and Atsushi Igarashi. Gradual typing for generics. In *Proceedings of the 26th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2011)*, pages 609–624, Portland, Oregon, USA, October 2011. ACM Press.

17 Nico Lehmann and Éric Tanter. Gradual refinement types. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*, pages 775–788, Paris, France, January 2017. ACM Press.

18 Ilya Sergey and Dave Clarke. Gradual ownership types. In Helmut Seidl, editor, *Proceedings of the 21st European Symposium on Programming Languages and Systems (ESOP 2012)*, volume 7211 of *Lecture Notes in Computer Science*, pages 579–599, Tallinn, Estonia, 2012. Springer-Verlag.

19 Jeremy Siek and Walid Taha. Gradual typing for functional languages. In *Proceedings of the Scheme and Functional Programming Workshop*, pages 81–92, September 2006.

20 Jeremy Siek and Walid Taha. Gradual typing for objects. In Erik 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 2007. Springer-Verlag.

21 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 274–293, Asilomar, California, USA, May 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

22 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. Monotonic references for efficient gradual typing. In Jan Vitek, editor, *Proceedings of the 24th European Symposium on Programming Languages and Systems (ESOP 2015)*, volume 9032 of *Lecture Notes in Computer Science*, pages 432–456, London, UK, March 2015. Springer-Verlag.

23 Matías Toro, Ronald Garcia, and Éric Tanter. Type-driven gradual security with references. *ACM Transactions on Programming Languages and Systems*, 40(4):16:1–16:55, November 2018.

24 Matías Toro, Elizabeth Labrada, and Éric Tanter. Gradual parametricity, revisited. *Proceedings of the ACM on Programming Languages*, 3(POPL):17:1–17:30, January 2019.

25 Matías Toro and Éric Tanter. A gradual interpretation of union types. In *Proceedings of the 24th Static Analysis Symposium (SAS 2017)*, volume 10422 of *Lecture Notes in Computer Science*, pages 382–404, New York City, NY, USA, August 2017. Springer-Verlag.

26 Niki Vazou, Éric Tanter, and David Van Horn. Gradual liquid type inference. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), November 2018.

27 Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In Mira Mezini, editor, *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP 2011)*, volume 6813 of *Lecture Notes in Computer Science*, pages 459–483, Lancaster, UK, July 2011. Springer-Verlag.

28 Ningning Xie, Xuan Bi, and Bruno C. d. S. Oliveira. Consistent subtyping for all. In Amal Ahmed, editor, *Proceedings of the 27th European Symposium on Programming Languages and Systems (ESOP 2018)*, volume 10801 of *Lecture Notes in Computer Science*, pages 3–30, Thessaloniki, Greece, April 2018. Springer-Verlag.