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 $\text{Int} \rightarrow \text{Bool}$ is more precise than $\text{?} \rightarrow \text{?}$, 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, $\text{?} \rightarrow \text{Int}$ is consistent with $\text{Bool} \rightarrow \text{?}$. 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. $\text{Ref Int}$), not whole types (e.g. $\text{Ref ?}$ is not supported). In this article we answer the following open questions:
Abstracting Gradual References

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 λfltREF, a gradual language with support for mutable references. We derive λfltREF by applying the AGT methodology to a fully-static simple language with mutable references called λREF. This is the first application of AGT that focuses on gradually-typed mutable references.
- We prove that λfltREF 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, λfltREF, corresponds to the semantics of guarded references from HCC. Formally, given a λfltREF 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 λfltREF and HCC+ differ in the order of combination of runtime checks. As a result, HCC is space efficient whereas λfltREF is not: we can write programs in λfltREF that may accumulate an unbounded number of checks. We formalize the changes needed in the dynamic semantics of λfltREF 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 λfltREF by presenting λpmREF, 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 λfltREF 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


Abstracting Gradual References


