An Implementation Strategy for Gradual Dependent Types

Joseph Eremondi
University of Edinburgh
United Kingdom
joey.eremondi@ed.ac.uk

1 Why Gradual Dependent Types?

Dependent type systems allow programmers to use types to express rich specifications along with proofs that the programs meet those specifications. However, actually writing programs that meet these specifications can be difficult.

Gradual dependent types introduce a degree of flexibility into a dependent type system. With gradual dependent types, parts of terms or types can be omitted. Types are compared for consistency relative to available information, and run-time checks are inserted to replace static checks deferred due to imprecision.

Gradual dependent types help with developing dependently typed programs in two ways. First, they provide a dynamic semantics for programs holes. Languages like Agda and Idris already allow parts of a type or proof to be omitted. Gradual dependent types go further by allowing such programs to be safely run. Second, gradual dependent types provide a smooth path for migrating non-dependently typed code to dependent languages, and to safely interacting with non-dependent code via an FFI.

Our hypothesis is that dynamic information from running a program is useful for constructing static proofs in dependently typed languages. However, to test this conjecture, we need an implementation of gradual dependent types.

2 Implementing Gradual Dependent Types

2.1 Implemenetation Challenges

There are many complex parts of a compiler for a dependently typed language. Because terms serve a double purpose as proofs, dependently typed programs often contain terms that serve no purpose at run time but exist to satisfy the type checker. So code practical generation requires heavy optimization.

Additionally, conversion checking is a major part of dependent type checking. Because types may depend on terms, comparing the equality of two types checks if they are convertible, i.e. have the same normal form. The naive approach of normalizing then comparing types is correct but slow, and efficient conversion checkers rely on many subtle tricks.

Implementing these tools is difficult and time consuming, even for non-gradual dependent types. With gradual dependent types, these tools must work in settings where precise types for terms may not be known, where termination is not guaranteed, and where dynamic checks must be added to ensure safety.

2.2 Implementation via Translation

We propose an implementation strategy for gradual dependent types where, instead of implementing a compiler directly, gradual dependent types are translated into static dependent types. Under this approach, a compiler for gradual dependent types can re-use conversion checking, optimization, and normalization from the static language. The translation is based on the syntactic model of Lennon-Bertrand et al. [5], extended with the techniques developed in [4]. A proof of concept for the translation has been written in Agda [1].

2.2.1 Termination. A major challenge with this approach is balancing the possibility for non-terimating gradual programs with the termination requirements of static dependent types systems. MLTT and CIC, upon which most proof assistants are based, both disallow non-terminating programs. Implementations of dependent types typically include "escape hatches" which can circumvent the termination checker, but these either allow for non-termination during type checking or limit the extent to which terms can be evaluated during conversion checking.

We instead follow the approach of [3], where conversion checks are performed on approximate versions of terms. These approximate terms lie in a terminating fragment of gradual dependent types, and hence can be embedded in a static dependently typed language without using any special features.

2.2.2 Equality. The second major challenge is finding a suitable gradual representation of propositional equality. The theory of gradual propositional equality is developed in [2], representing equality proofs between two terms with witnesses as precise as those two terms. However, this introduces a mutual dependency between the operations that convert between types and compose run-time type information. In order to represent gradual equality proofs, our translation to static types relies on a bespoke library of ordinals to prove these operations terminate.

References

[1] Joseph Eremondi. 2023. Github Repository: Guarded Model of Gradual Dependent Types. https://github.com/JoeyEremondi/GuardedModel/tree/externalReview/model.

[2] Joseph Eremondi, Ronald Garcia, and Éric Tanter. 2022. Propositional Equality for Gradual Dependently Typed Programming. Proc. ACM Program. Lang. 6, ICFP, Article 96 (August 2022), 29 pages. https://doi.org/10.1145/3547627

[3] Joseph Eremondi, Éric Tanter, and Ronald Garcia. 2019. Approximate Normalization for Gradual Dependent Types. Proc. ACM Program. Lang. 3, ICFP, Article 88 (July 2019), 30 pages. https://doi.org/10.1145/3341692

[4] Joseph S. Eremondi. 2023. On the design of a gradual dependently typed language for programming. Ph. D. Dissertation. University of British Columbia. https://doi.org/10.14288/1.0428823

[5] Meven Lennon-Bertrand, Kenji Maillard, Nicolas Tabareau, and Éric Tanter. 2022. Gradualizing the Calculus of Inductive Constructions. ACM Trans. Program. Lang. Syst. 44, 2, Article 7 (apr 2022), 82 pages. https://doi.org/10.1145/3495528