A Guraded Syntactic Model of Gradual Dependent Types

Translation to Support Implementation and Metatheory

ANONYMOUS AUTHOR(S)

TODO write abstract

1 INTRODUCTION

Gradual dependent give principled static and dynamic semantics to programs where part of a type or term is missing. By allowing imprecision, gradual dependent types allow for smooth migration of code from non-dependently typed languages, or even untyped languages, to full dependent types, allowing the programmer run and test their code even when the full type details haven't been figured out. This migration is easiest in languages fulfilling the *gradual guarantees* of Siek et al. [2015], which state replacing part of a program with ? creates no new static or dynamic type errors. The gradual guarantees ensure that, when an error is encountered, the problem is never too few types, but that two types in the program are fundamentally incompatible.

However, the benefits of gradual dependent types have not been realized, since existing developments have enabled the gradual guarantees at the expense of other desirable properties. Eremondi et al. [2019] presented GDTL, a dependent calculus supporting the gradual guarantees, but relied on a termination argument that does not scale to inductive types. Lennon-Bertrand et al. [2022] presented GCIC, with two extensions of the Calculus of Inductive Constructions (CIC) that satisfy the gradual guarantees, but one has undecidable type checking and the other rejects some well-typed static CIC programs.

Lennon-Bertrand et al. [2022] show that, to a degree, these sacrifices are unavoidable, to a degree: no dependently typed language can satisfy all of strong normalization, conservative extension of CIC, and *graduality*, a strengthening of the gradual guarantees.

Another obstacle to the adoption of gradual dependent types is that gradual dependent types have not yet been meaningfully implemented. Constructing a compiler for a dependently typed language is a massive engineering effort, and involves writing a type checker, a convertability check for terms, and unification engine for inference, in addition to the code generation and optimization. Writing a compiler for a gradual dependently typed language involves all of this work, plus extra handling to ensure safety in the presence of type imprecision.

We address both these shortcomings in GrInd, a Gradual language with Inductive types. GrInd sacrifices strong normalization and graduality, but keeps the properties that we actually want: decidable type checking, (weak) consistency and canonicity, the gradual guarantees, and conservatively extending CIC. Because type checking is decidable, GrInd can be translated into the core calculi of existing dependently typed compilers. Moreover, we show that GrInd does not violate static reasoning principles: propositionally-equal CIC terms embedded in GrInd are observationally-equivalent, and casts only change the error-behavior of terms, not the concrete results produced, a notion which we call *weak graduality*.

Our main contribution is a translation from GrInd to a static type theory:

- For implementation, the translation means that existing normalizers and code generators can be used "off-the-shelf" to compile GrInd programs;
- For metatheory, the translation serves as a syntactic model in the style of Boulier et al. [2017], which we use to prove the gradual guarantees and other metatheoretic properties;
- To enable decidable type checking, we adapt approximate normalization from Eremondi et al. [2019] to a cast calculus, using the syntactic model to prove termination;

- To model run-time semantics, we translate to *guarded type theory* [?], whose non-positive recursive types allow the non-termination of gradual types to be exactly represented in a consistent target language;
- Our translation and the theorems about it have been mechanized in Guarded Cubical Agda [?]

2 TWO PROBLEMS, ONE SOLUTION

Our work attacks two main problems that have a common solution. First, we want to allow GrInd to be implemented by translating them to static dependent types without needing to add features to the static target langauge, so that existing technology can be used when compiling them. The challenge here is accommodating the non-termination of gradual typing, since dependently typed core languages typically forbid or restrict non-terminating function definitions. Second, we want to prove properties about GrInd, namely that approximate normalization terminates (for decidable type checking) and that the gradual guarantees are satisfied. Unlike the approach of Eremondi et al. [2019], the syntactic model approach scales to handle inductive types, as well as logical-relation style proofs [Bernardy et al. 2012].

In this section, we explain these two problems, the challenges in solving them, and a birds-eye view of our approach to solving them.

Throughout, we refer to terms from several languages. For clarity, we write terms from CIC, the static starting language, in red sans-serif font, terms from GrInd, the gradual cast calculus, in **blue, bold serif font**, and terms from the guarded type theory, the target of translation, in *green, italic serif font*.

2.1 Translation to Support Implementation

2.1.1 Don't Reinvent the Wheel. Even without gradual types, implementing a dependently typed language is a formidable task. Because types may depend on terms, checking that two types are equal involves determining if those two types *convertible*, i.e., if they are equal after some number of reductions. Such a check must be implemented carefully, usually using Normalization by Evaluation (NbE) [?], to ensure that only well-typed terms are normalized so the check terminates. Support for type inference and implicit arguments to functions involves a writing higher-order unification algorithm. Most dependently typed languages have some form of proof-search, tactics, or reflection, to automate the generation of trivial but tedious proofs.

Even with the above features implemented, implementing them efficiently is even more difficult. Conversion checking is a costly operation, and while it can be implemented by fully normalizing the compared terms, this likely performs more computation than is necessary to see whether the terms are definitely convertible or not. Recent experiments in Idris have shown that on-the-fly compilation of terms can provide efficient normalization [?]. For code generation, since the return type of a function can depend on the value of its argument, the generation of machine code more closely resembles a dynamic language, but from an optimization standpoint, programs contain a wealth of type information that can be used to optimize said dynamic code. Moreover, dependently typed programs may contain indices or function arguments that are needed only because of type dependency, but are unused during run time [?]. Additional optimizations are needed to identify and remove unused terms from the generated code. Gradual dependent types should support the latest and greatest techniques for efficiency in both the compiler and compiled code.

2.1.2 An Implementation Strategy. To avoid re-implementing the above tools for gradual dependent types, our implementation strategy is to first translate gradual dependent programs to a static dependently typed language. Then, existing implementations of conversion checking, unification,

optimization and code-generation can be freely used on the result of translation. A gradual dependently typed language can simply be a new front-end to an existing dependently typed language, such as Agda or Idris.

The challenge with this approach is that, to reuse existing infrastructure, we must be able to translate gradual dependent types to static dependent types *without changing the static core language*. Gradual dependently typed programs do not always terminate, and may raise run-time type errors. However, the termination of static tools relies on the termination of static programs. Escape hatches exist, like Agda's `{-# TERMINATING #-}` pragma, but using such features is morally wrong, as these features are intended for cases where programs do terminate, but their termination is difficult or impossible to prove within the type theory itself. Instead, we must work within more conservative features for taming non-terminaton.

2.1.3 Idris and Agda's Models of Non-Termination. In Idris, all definitions are marked as either `partial` or `total`. Those that are marked `total` can only refer to others that are marked `total`. Recursive `total` functions are checked to ensure that recursive calls are only made to smaller arguments, and `total` inductive datatypes must be strictly positive (i.e. self-reference may only occur) to the right of function arrows in fields). By contrast, `partial` functions allow for general recursion and non-positive datatypes, and

The key to keeping type-checking decidable is to never normalize `partial` terms. In Idris, when a type depends on a `partial` function, the definition of that function is kept abstract.[1] This convention allows for trivial equalities to be satisfied, like a term being equal to itself. Two terminating terms that are marked `partial` are not considered definitionally equal even if they have the same evaluation, because normalization is required to see that they are equal. Agda has a similar model: definitions may be marked as `{-# NON_TERMINATING #-}`, which allows for general recursion but prevents such definitions from reducing during type checking.

The issue with these models is that they are difficult to reason about in the abstract. Running the type-checker immediately reveals whether a particular definition type checks with `partial` annotations. However, we wish to prove that our gradual to static translation produces well typed code for *all* outputs, but doing so requires reasoning about the internal details of Idris or Agda's normalization algorithm.

To simplify the proof of our translation's type preservation, we instead turn to *guarded type theory*. We explain more details of guarded type theory in §2.3.2, but for now it suffices to say that guarded type theory allows for a restricted form of general recursion and non-positive datatypes, by placing such self-reference behind a separate modality, and establishing fixed-point equalities as propositional equalities, rather than definitional equalities. Guarded type theory provides a convenient formalism for reasoning about when definitions are normalized, and allows us to mechanize our translation in Guarded Cubical Agda **?**. Then, the primitives of guarded type theory can be easily implemented using the less restrictive `partial` annotations of Agda and Idris, allowing us to prove that the resulting code is always well typed.

2.1.4 Translating Approximate Normalization. While guarded type theory allows us to represent non-terminating terms at the type level, it provides no clues as to actually devise a gradual type system for which checking is decidable. However, previous work on gradual dependent types does provide clues. Lennon-Bertrand et al. [2022] were able to provide a syntactic model of GCIC without

[1] In Idris 1, this behaviour was implemented in the compiler. Idris 2 has yet to implement this behavior, and happily runs the type checker forever when comparing non-terminating terms. However, safety can be recovered by defining partial functions in their own module, and exposing them with `export` rather than `public export`, which reveals their types while keeping their definitions abstract.

decidable type-checking by extending CIC with a non-positive datatypes, then translating the unknown type ? to a non-positive inductive type $Unk := (Unk \to Unk) + (Unk \times Unk) + (List\ Unk) \ldots$. Eremondi et al. [2019] were able to obtain terminating type level computations in GDTL with *approximate normalization*: terms occurring within types were given a separate semantics that produced ? whenever there was not enough type information to ensure termination.

We combine these two strategies to obtain a translation with decidable type checking. Like GDTL, we have separate semantics for compile-time normalization and run-time execution. However, GDTL had totally separate approximate and exact semantics, where approximation was based on hereditary substitution, no formal notion related the two semantics. Our approximate and exact semantics are identical except for one part: casts between ? and $? \to ?$. Instead of relying on hereditary substitution, we use the GCIC approach and translate it to Martin-Löf Type Theory: the approximations mean that only strictly-positive datatypes are used, and hence termination is guaranteed.

2.2 Metatheory

2.2.1 *Extinguishing the Fire Triangle.*

2.2.2 *Static Reasoning in Gradual Code.*

2.3 Modelling Gradual Dependent Types

2.3.1 *Modelling Approximate Normalization.*

2.3.2 *Guarded Type Theory.*

2.3.3 *Relating Approximate and Exact Normalization.*

REFERENCES

Jean-philippe Bernardy, Patrik Jansson, and Ross Paterson. 2012. Proofs for Free: Parametricity for Dependent Types. *J. Funct. Program.* 22, 2 (mar 2012), 107–152. https://doi.org/10.1017/S0956796812000056

Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The next 700 Syntactical Models of Type Theory. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs* (Paris, France) *(CPP 2017)*. Association for Computing Machinery, New York, NY, USA, 182–194. https://doi.org/10.1145/3018610.3018620

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

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

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Vol. 32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274