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) 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*. To avoid confusion, and to emphasize GrInd terms as syntax but terms from guarded type theory as mathematical entities, we write the type of types, dependent function types, and dependend pair types as $\mathbf{Type}_\ell$, $(\mathbf{x} : \mathbf{T}_1) \to \mathbf{T}_2$ and $(\mathbf{x} : \mathbf{T}_1) \times \mathbf{T}_2$ for GrInd, but use $\mathcal{U}_\ell$, $\Pi(x : T_1).\ T_2$ and $\Sigma(x : T_1).\ T_2$.

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

[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.

Language	Decidable type checking	Gradual Guarantees	Conservative Over CIC	Weak Graduality	Weak Canonicity	Inductives	Graduality	Strong Normal-ization
GDTL	✓	✗*	✗	?	✓	✗	✗	✗
$GCIC^{\mathcal{G}}$	✗	✓	✓	✓	✓	✓	✓	✗
$GCIC^{\mathcal{N}}$	✓	✗	✓	✗	✓	✓	✗	✓
$GCIC^{\uparrow}$	✓	✓	✗	✓	✓	✓	✓	✓
GrIndd	✓	✓	✓	✓	✓	✓	✗	✗

Table 1. Gradual Dependent Languages: Feature Comparison

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 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, GrIndd has 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 CIC: 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. When developing GCIC, Lennon-Bertrand et al. [2022] establish a "fire-triangle" theorem stating that no language could conservatively extend CIC while having strong normalization and graduality. Here, graduality is the property defined by New and Ahmed [2018], where casting to a more precise type and back produces a term that is as precise as the original, and casting to a less precise type and back produces an observationally-equivalent term. The name is chosen by analogy with parametricity: just as parametricity is a logical relation strengthening type safety, graduality is a logical relation strengthening the gradual guarantees. GCIC is then presented with three variants: one without decidable type checking, one without the gradual guarantees, and one that rejects some well-typed CIC programs.

In this paper, we take the view that for programming languages specifically, strong normalization and graduality are means to an end, rather than ends unto themselves. Specifically, we want strong normalization to prove decidable type checking and weak canonicity i.e. that all normal-forms of a term are either canonical, ?, or an error $\mathcal{U}$. Strong normalization is also required for weak logical consistency: GrIndd does not have this, but we discuss how to recover a form of it in §2.2.4. We want graduality to prove the gradual guarantees, as well as to establish that ? is never needlessly produced as a result. (Such a property is desirable because a language that produces ? as a result of all casts is useless, but still technically satisfies the gradual guarantees.)

So while GrIndd cannot simultaneously have all three fire-triangle properties, it can have all the properties we actually want: decidable type checking, the gradual guarantees, and conservatively extending CIC. Section 2.2.1 compares the features of GDTL, the three variants of GCIC, and GrIndd: while GrIndd sacrifices strong normalization and graduality, we are able to recover all the other desired properties despite this.

However, proving these properties is not trivial. GDTL proved decidability of type checking using hereditary substitution, which does not scale well to inductive types. The gradual guarantees were proved using a simulation argument, but the proof was complex and delicate. While a syntactic model was used, the $GCIC^{\mathcal{G}}$ variant that embeds CIC and has the gradual guarantees required non-positive datatypes, so the model was by a translation to an inconsistent logic, and the target type theory could not be used to prove theorems about $GCIC^{\mathcal{G}}$. For the other variants, strong normalization was shown in a syntactic model by sacrificing either embedding CIC or the gradual guarantees.

Our implementation strategy works equally well for proving that type checking is decidable: if we can translate approximate normalization into the total fragment of a type theory, then it must be terminating. Then, by translating our exact semantics into guarded type theory, we are able to prove the gradual guarantee in the type theory itself. Unlike CIC with non-positive types, guarded type theory is logically consistent, so the theorems proved about the translations of gradual programs can be believed.

2.2.2 Preservation of Propositional Equality. We show that, for any two static CIC terms $\mathrm{s}, \mathrm{t} : \mathrm{T}$, if we have a proof that $\mathrm{s} =_{\mathrm{T}} \mathrm{t}$, then when we embed s and t into GrInd, the resulting $\mathbf{s}$ and $\mathbf{t}$ are observationally equivalent. This guarantees that static reasoning principles still hold for fully static code that is used in gradual programs. For example, any optimizations that are valid for static code are still valid if that code is imported by a gradual library.

The preservation of equality is a weaker version of the gradual full-abstraction property described by Jacobs et al. [2021], which says that all observationally-equivalent static programs are observationally-equivalent when embedded in the gradual language. Full abstraction is difficult to prove for GrInd, since the conventional strategy requires the ability to translate any GrInd context into an equivalent CIC one, which is impossible since CIC lacks non-termination as an effect. Proving that propositionally-equal static terms are gradually-equivalent provides a good first step, however, to ensuring that static is preserved gradually. JE ▶*TODO: check if this prevents the bad example from DeVrise paper*◀ JE ▶*TODO: can we prove FA with funExt, by turning* $(x : T)$ *into* $\lambda(C : T \to B).\, Cx$ *and using fun-ext?* ◀ JE ▶*TODO: can we prove FA from CIC->GuardedTT FA?*◀

2.2.3 Weak Graduality. Weak graduality: while we do not prove the strong graduality property of New and Ahmed [2018], we can prove a weaker version that establishes the same desirable property.

JE ▶*TODO: should we cut all this and just to EP-pairs? We'll see how complicated everything else is*◀ Why does GrInd not satisfy strong graduality? In principle, it is possible to prove strong graduality for a variant of GrInd that ensured that approximate normal forms were never used at run-time, since the approximation when converting functions to and from $\mathbf{?}$ is what breaks the property. However, such a language would introduce the possibility that computations internal to run-time type information could non-terminate. These would be exceedingly difficult for the programmer to debug, since such computations are automatically inserted by the compiler and are meant to be opaque to the programmer. This is not an issue for non-dependent gradual types, where types do not contain terms and there is no possibility for non-termination, but in the dependent setting, it is a key usability issue.

We formalize this property with a novel criterion that we call the *gradual termination guarantee:*

Definition 2.1 (The Gradual Termination Guarantee). For a gradual surface term t, if $\Gamma \vdash t : T_1$ and $\Gamma \vdash t : T_2$, then the elaboration of t at T_1 terminates iff the elaboration of t at T_2 terminates.

Equivalently, for a cast calculus term $\mathbf{t}$, if $\Gamma \vdash \mathbf{t} : \mathbf{T}$, then for all $\mathbf{T'}$, $\langle \mathbf{T'} \Leftarrow \mathbf{T} \rangle \mathbf{t}$ terminates iff $\mathbf{t}$ terminates.

In essence, the termination guarantee says that the termination behavior of a term depends only on the term itself, not on its type. The termination guarantee is incompatible with strong graduality: casting to a less precise type then back must produce an equivalent term under strong graduality, so no approximations can be used, even in type ascriptions, leaving the possibility for non-termination in type information.

What we need, then, is a way to establish that Grlnd does not needlessly produce $\mathbf{?}$ as the result of run-time casts while still preserving the termination guarantee. To do this, we define *weak graduality* to be as follows:

Definition 2.2 (Weak Graduality). If $\mathbf{T}_1 \sqsubseteq \mathbf{T}_2$ (i.e. $\mathbf{T}_1$ is more precise than $\mathbf{T}_2$), then:

- For any $\mathbf{t}_2 : \mathbf{T}_2$ and context $C : \mathbf{T}_2 \to \mathbb{B}$, if $C[\langle \mathbf{T}_2 \Leftarrow \mathbf{T}_1 \rangle \langle \mathbf{T}_1 \Leftarrow \mathbf{T}_2 \rangle \mathbf{t}_2] \longrightarrow^* \mathbf{v}_1$ and $C[\mathbf{t}_2] \longrightarrow^* \mathbf{v}_2$, then $\mathbf{v}_1 \sqsubseteq_\alpha \mathbf{v}_2$.
- For any $\mathbf{t}_1 : \mathbf{T}_1$ and context $C : \mathbf{T}_1 \to \mathbb{B}$, if $C[\mathbf{t}_1] \longrightarrow^* \mathbf{v}_1$ and $C[\langle \mathbf{T}_1 \Leftarrow \mathbf{T}_2 \rangle \langle \mathbf{T}_2 \Leftarrow \mathbf{T}_1 \rangle \mathbf{t}_1] \longrightarrow^* \mathbf{v}_2$, then either $\mathbf{v}_1 = \mathbf{\mathfrak{U}}$ or $\mathbf{v}_1 = \mathbf{v}_2$.

Unlike strong graduality, the second criterion does *not* state that the terms are observationally equivalent: the result of casting may cause an error in fewer contexts than the original term. However, it does say that, when neither term errors, the results are identical. No context exists such that a round-trip cast will produce $\mathbf{?}$ but the original term produces $\mathbf{true}$ or $\mathbf{false}$. The difference is only meaningful because we allow $\mathbf{?}$ as a term in gradual dependent types; for non-dependent types the definition collapses to error-approximation [New and Ahmed 2018].

The proof of weak graduality is obtained using a logical-relations style proof, and the syntactic-model directly enables this, allowing us to devise a logical relation in the style of Bernardy et al. [2012]. Moreover, the gradual guarantees follow from weak graduality, avoiding the need for GCIC-style simulation arguments.

2.2.4 Weak Approximate Consistency. With approximate normalization we recover a form of weak logical consistency: any closed term of type $\mathbf{False}$ must have an approximate normal form that is either the unknown term $\mathbf{?}$ or a run-time type error $\mathbf{\mathfrak{U}}$. This property is useful for extrinsic proofs that are not used in computation, but serve as validation that some desired property holds. Because some Grlnd programs are non-terminating, it is possible to write closed terms of statically-empty types that do not evaluate to $\mathbf{?}$ or $\mathbf{\mathfrak{U}}$. However, inspecting the approximate-normal forms of such proofs can inform the programmer about places where type-imprecision caused approximation, directing them to add type information to refine the result. If a term of a fully-static type has a fully-static approximate normal form, then the programmer can be confident in the truth of its Curry-Howard interpretation, even if the proof itself uses imprecise types or terms.

2.3 Modelling Gradual Dependent Types

With a handle on the two problems we are trying to solve by translating gradual dependent types to static, we next give an overview of how the translation works.

2.3.1 Modelling Approximate Normalization. For approximate normalization, the main goal is showing that it always terminates. So to do this, we translate each Grlnd term to a term in CIC. The difference between CIC and MLTT is not overly significant for us: we use only the predicative fragment of CIC and avoid cumulativity, JE ►*TODO do we?*◄ , so the main difference is the use of an untyped reduction relation instead of a typed conversion judgment. This reduction relation is convenient for establishing a simulation between Grlnd's reduction rules and CIC's reductions, so we opt for CIC. Functions are tagged with a superscript ^{A} to denote that they are for the Approximate model. JE ►*TODO: do we need this, if we have the back-translation?*◄

The translation itself works like the discrete model of GCIC [Lennon-Bertrand et al. 2022], differing only in the representation of $\mathbf{?}$ and casts to $\mathbf{?}$.

- Each datatype $\mathbf{D}$ is translated to a datatype D with two additional constructors, denoting $\mathbf{?} : \mathbf{D}$ and $\mathfrak{U} : \mathbf{D}$ respectively;
- To keep run-time type information, we define a Universe à la Tarski $\mathbb{C}^A{}_\ell : \mathcal{U}$, with an "elements-of" interpretation function $\mathbf{El}^A{}_\ell : \mathbb{C}^A{}_\ell \to \mathcal{U}$;
-
- $\mathbf{?} : \mathbf{T}$ and $\mathfrak{U} : \mathbf{T}$ are translated via functions $\top$ and $\bot$, each with type $\Pi(c : \mathbb{C}^A{}_\ell).\ (\mathbf{El}^A{}_\ell\ c)$
- $\mathbf{?} : \mathbf{Type}_\ell$ is translated into an inductive type $Unk^A := 1 + 1 + \mathbb{C}^A + (Unit \to Unk^A) + (Unk^A \times Unk^A) + (1$ The first two entries are tags denoting $\mathbf{?} : \mathbf{?}$ and $\mathfrak{U} : \mathbf{?}$. To keep the type strictly positive, functions have an entry as $\mathbf{Unit} \to \mathbf{Unk}$ rather than $Unk \to Unk$;
- Casts are translated via a function $cast : \Pi(c_1 : \mathbb{C}^A\ell)(c_2 : \mathbb{C}^A\ell).\ \mathbf{El}^A\ c_1 \to \mathbf{El}^A\ c_2$;
- A cast $\langle \mathbf{A} \Leftarrow^{\rangle} \mathbf{?} \to \mathbf{??}$ is translated to a function that transforms $f : Unk \to Unk$ into $\lambda\_.\ (f\ (\top\ C?))$, where $\mathbf{El}^A{}_\ell\ C? = Unk$, so that the result can be injected into Unk. We prove that all GrInd functions have precision-monotone translations, $f\ (\top\ C?)$ is the least precise value produced by f.

Our method of approximating a function by applying it to $\top\ C?$ is actually more precise than the approximation performed in GDTL [Eremondi et al. 2019], which simply replaced the function with $\mathbf{?}$. This is especially convenient since constant functions are not approximated at all. Constant functions arise frequently in dependently typed programming, when functions with dependent types are used in non-dependent cases. JE ►*TODO explain more?*◄

2.3.2 Guarded Type Theory and Non-Positive Types. To understand how we make a syntactic model of exact execution, we first introduce guarded type theory and its primitives. Guarded type theory comes in many varieties [Bahr et al. 2017; Birkedal et al. 2011; Birkedal and Møgelberg 2013; Gratzer and Birkedal 2022; Sterling and Harper 2018], which vary primarily on which equalities are provable in them. We avoid picking a particular theory, since we only need a few sets of primitives. Our model is mechanized in Guarded Cubical Agda [Veltri and Vezzosi 2020], which is based on Ticked Cubical Type Theory [Møgelberg and Veltri 2019]. However, we only use cubical features to prove metatheory: the model itself requires only the basic primitives of CIC and the features we discuss below.

At the heart of guarded type theory is the *later modality* $\triangleright : \mathcal{U}_\ell \to \mathcal{U}_\ell$. Intuitively, $\triangleright T$ denotes an element of T that is not available now, but will be available one time-step in the future. The modality $\triangleright$ is an applicative functor: there are operators $next : T \to \triangleright T$ and $\_\circledast\_ : \triangleright(A \to B) \to \triangleright A \to \triangleright B$ that allow curried functions to be mapped over guarded arguments.

The later modality gives a sound way to take arbitrary fixed-points: guarded type theory features a constant $fix : (\triangleright A \to A) \to A$, along with a propositional equality $löb : fix\ f =_A f\ (next\ (fix f))$, named after Löb induction [?]. This allows for general recursion, provided that

How are guarded fixed-points useful, given that the guarded modality is "inescapable" and there is no general way to go from $\triangleright A$ to A? The key is *guarded recursive types*. Guarded type theory has a guarded type operator $\blacktriangleright : \triangleright\mathcal{U}_\ell \to \mathcal{U}_\ell$, which transforms a guarded reference to a type into a type. Using this, one can construct a non-positive type fixed-point $tyfix : (\mathcal{U}_\ell \to \mathcal{U}_\ell) \to \mathcal{U}_\ell$, such that $tyfix\ F =_{\mathcal{U}_\ell} F\ (\triangleright(tyfix\ F))$. Essentially, $tyfix$ allows for non-positive inductive types, but requires all negative self-reference to be guarded. Then fix and $löb$ can be used to perform induction over types formed with $tyfix$, provided the results from recursive calls on guarded children are of a guarded result type.

With this, we have (nearly) enough to model $?$ exactly as a guarded recursive type: $Unk = tyfix(\lambda X.\ Unit + Unit + \mathbf{Code}^E{}_\ell + (X \to Unk) + (Unk \times Unk) + List\ Unk \ldots)$. However, we still need a way to include possibly non-terminating functions in Unk, which we explain below.

2.3.3 The Lifting Monad and Modelling Non-Termination. The later modality denotes a value available one step in the future, but to fully model non-terminating computation, we need a way to represent values available an arbitrary number of steps into the future. Paviotti et al. [2015] describe how to do this with a guarded "lifting monad" $L\ A = A + \triangleright(L\ A)$. An $L\ A$ value is either available now, or is another $L\ A$ value available one step into the future. The usual monadic *pure* and *bind* operations are available on L. Critically, one can write an operation $\theta : \triangleright(L\ A) \to L\ A$ that lets a guarded value be injected into the monad, meaning we can use *fix* to write $fixL : (L\ A \to L\ A) \to L\ A$: general recursion is allowed so long as the result is in the lifting monad.

This gives us the correct representation of $?$ in our syntactic model:

$$Unk^E = tyfix(\lambda X.\ Unit + Unit + \mathbf{Code}^E{}_\ell + (X \to L\ Unk^E) + (Unk^E \times Unk^E) + List\ Unk^E \ldots)$$

Casting $f : Unk^E \to Unk^E$ to Unk^E can be done by injecting $(\lambda(x : \triangleright Unk^E).\ \theta((next\ f) \circledast x))$ into Unk^E.

To properly model casts out of Unk^E, we need one final primitive, a transformer $liftFun' : (A \to L\ B) \to \triangleright(L\ (A \to B))$. This can be implemented directly in ticked cubical type theory, and we imagine that it is expressible in other guarded theories. JE ▶*TODO: other guarded theories? Check*◀ With this primitive, we can combine it with θ to obtain $liftFun : (A \to L\ B) \to (L\ (A \to B))$. With this primitive, casts can be implemented as a function producing a possibly non-terminating result (via the lifting monad:)

$$cast^E : \Pi(c_1 : \mathbf{Code}^E{}_\ell)\Pi(c_2 : \mathbf{Code}^E{}_\ell).\ \mathbf{El}^E\ c_1 \to L\ (\mathbf{El}^E\ c_2)$$

That is, casting takes two codes describing types and an element of the first type, and produces a computation that, if it terminates, contains an element of the second code's type. For example, to cast from Unk^E to $Unk^E \to Unk^E$, a function $f : \triangleright Unk^E \to L(Unk^E)$ can be transformed into $liftFun\ (\lambda(x : Unk^E).\ f(next\ x))$, which has type $L\ (Unk^E \to Unk^E)$.

Using *liftFun* lets us separate values from computations: for each code $c : \mathbf{Code}^E{}_\ell$, its interpretation $\mathbf{El}^E\ c$ is *not* under the lifting monad. Elimination forms are translated into applications of functions that return results under L, capturing how computation may diverge.

2.3.4 Relating Approximate and Exact Normalization. So far, we have given an overview of how to translate types and computation both approximately and exactly. However, the two must be related to produce an accurate account of GrIndʼs type system. What we have so far can only apply dependently typed functions to values, not to the results of computation.

To see the issue, consider the function $\mathbf{repeatFalse} : (\mathbf{n} : \mathbb{N}) \to \mathbf{Vec}\ \mathbb{B}\ \mathbf{n}$, which takes an element and creates a vector that has $\mathbf{false}$ repeated $\mathbf{n}$ times. This would be translated into $repeatFalse : L\ ((n : \mathbf{El}^E\ CNat)$ where $CNat$, $CBool$ and $CVec$ are the code constructors for natural numbers, booleans, and vectors respectively. The question is then, how should $\mathbf{repeatFalse}\ (\langle \mathbb{N} \Leftarrow ? \rangle \mathbf{t})$ be translated for some $\mathbf{t} : ?$? In our model, *cast* produces a value in the lifting monad L, but we need an un-lifted member of $El^E\ CNat$ to compute the return type. Using the monadic bind allows this, but produces a lifted type, i.e., $bind(cast \ldots)(\lambda n \to \mathbf{El}^{EE}\ (CVec\ CBool\ n)) : L\ \mathcal{U}$.

A potential option is to implement an operation $unLiftType : L\ \mathcal{U}_\ell \to \mathcal{U}_\ell$ using $\blacktriangleright$. The problem then is that, in general, we can only convert between $unLiftType\ S$ to $\mathbf{El}^E\ c$ when S terminates and we can statically see that they denote the equal (or at least isomorphic) types. Such a translation works fine for modelling $GCIC^{\mathcal{G}}$, where type checking is undecidable and programs are considered

ill-typed if the conversion check never terminates. However, in GrInd, the conversion check for sufrace typing looks like this:

$$\frac{\Gamma \vdash t \Rightarrow \mathbf{T}_2 \qquad \mathbf{T}_1 \longrightarrow^*_{Approx} \mathbf{T}_1' \qquad \mathbf{T}_2 \longrightarrow^*_{Approx} \mathbf{T}_2' \qquad \mathbf{T}_1' \cong \mathbf{T}_2'}{\Gamma \vdash t \Leftarrow \mathbf{T}_1}$$

A surface term t checks against a type $\mathbf{T}_1$ if t synthesizes a type $\mathbf{T}_2$ that is consistent with $\mathbf{T}_1$ after some number of reductions *according to the approximate semantics*. This is necessary to keep type checking decidable. But then, there are programs that will be well-typed in GrInd but ill-typed in the model because S does not terminate and hence $unLiftType\ S$ is a guarded type.

JE ►*TODO: concrete example*◄

Instead, our solution is to ensure that, in the model, dependent types only ever depend on the *approximate normal forms* of terms. We define $\mathbf{Code}^E = \mathbb{C}^A$, and translate each GrInd term $\mathbf{t} : \mathbf{T}$ into a pair with type $(\mathbf{El}^A\ c) \times L\ (\mathbf{El}^E\ c)$ i.e. an approximation paired with an exact but possibly non-terminating computation. Then, the inability to convert from $L\ \mathcal{U}_\ell$ to $\mathcal{U}_\ell$ is a feature, rather than a bug: it ensures that types only depend on the approximate form of computations, ensuring that the model faithfully represents GrInd's typing. For dependent functions, there is a constructor $CPi : (dom : \mathbb{C}^A) \to (\mathbf{El}^A\ dom \to \mathbb{C}^A) \to aCode$, with $\mathbf{El}^E\ (CPi\ dom\ cod) = \Pi(x : (\mathbf{El}^A\ dom) \times L\ (\mathbf{El}^E\ dom)).\ \mathbf{El}^E\ (cod\ (fst\ x))$. So the return type of a function only ever depends on the approximate normal form of the argument. JE ►*TODO: check this, this is the idea but might not be exactly how El is defined*◄

REFERENCES

Patrick Bahr, Hans Bugge Grathwohl, and Rasmus Ejlers Møgelberg. 2017. The clocks are ticking: No more delays!. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–12. https://doi.org/10.1109/LICS.2017.8005097

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

L. Birkedal, R. E. Mogelberg, J. Schwinghammer, and K. Stovring. 2011. First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. 55–64. https://doi.org/10.1109/LICS.2011.16

L. Birkedal and R. E. Møgelberg. 2013. Intensional Type Theory with Guarded Recursive Types qua Fixed Points on Universes. In *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*. 213–222. https://doi.org/10.1109/LICS.2013.27

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

Daniel Gratzer and Lars Birkedal. 2022. A stratified approach to Löb induction. *Formal Structures for Computation and Deduction (to appear)* (2022). https://doi.org/10.4230/LIPIcs.FSCD.2022.3

Koen Jacobs, Amin Timany, and Dominique Devriese. 2021. Fully Abstract from Static to Gradual. *Proc. ACM Program. Lang.* 5, POPL, Article 7 (Jan. 2021), 30 pages. https://doi.org/10.1145/3434288

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

Rasmus Ejlers Møgelberg and Niccolò Veltri. 2019. Bisimulation as Path Type for Guarded Recursive Types. *Proc. ACM Program. Lang.* 3, POPL, Article 4 (jan 2019), 29 pages. https://doi.org/10.1145/3290317

Max S. New and Amal Ahmed. 2018. Graduality from Embedding-Projection Pairs. *Proc. ACM Program. Lang.* 2, ICFP, Article 73 (July 2018), 30 pages. https://doi.org/10.1145/3236768

Marco Paviotti, Rasmus Ejlers Møgelberg, and Lars Birkedal. 2015. A Model of PCF in Guarded Type Theory. *Electron. Notes Theor. Comput. Sci.* 319, C (dec 2015), 333–349. https://doi.org/10.1016/j.entcs.2015.12.020

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

Jonathan Sterling and Robert Harper. 2018. Guarded Computational Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 879–888. https://doi.org/10.1145/3209108.3209153

Niccolò Veltri and Andrea Vezzosi. 2020. Formalizing π-Calculus in Guarded Cubical Agda. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 270–283. https://doi.org/10.1145/3372885.3373814