Approximate Normalization for Gradual Dependently-Typed Languages

JOSEPH EREMONDI, University of British Columbia, Canada
RONALD GARCIA, University of British Columbia, Canada
ÉRIC TANTER, University of Chile, Chile

Dependent types help programmers write highly reliable code. However, this reliability comes at a cost: it can be challenging to write new prototypes in (or migrate old code to) dependently-typed programming languages. Gradual typing was developed specifically to make static type disciplines more flexible, so an appropriate notion of gradual dependent types could fruitfully lower this cost. However, dependent types raise unique challenges for gradual typing. Dependent typechecking involves the execution of program code, but gradually-typed code can signal runtime type errors or diverge. These runtime errors threaten the soundness guarantees that make dependent types so attractive, while divergence spoils the type-driven programming experience.

This paper presents GDTL, a gradual dependently-typed language, with an emphasis on pragmatic dependently-typed programming. GDTL fully embeds both an untyped and dependently-typed language, and allows for smooth transitions between the two. In addition to gradual types we introduce *gradual terms*, which allow the user to be imprecise in type indices and to omit proof terms; runtime checks ensure type soundness. To account for nontermination and failure, we distinguish between compile-time normalization and run-time execution: compile-time normalization is *approximate* but total, while runtime execution is *exact*, but may fail or diverge. We prove that GDTL has decidable typechecking and satisfies all the expected properties of gradual languages. In particular, GDTL satisfies the static and dynamic gradual guarantees: reducing type precision preserves typedness, and altering type precision does not change program behavior outside of dynamic type failures. To prove these properties, we were led to establish a novel *normalization gradual guarantee* that captures the monotonicity of approximate normalization with respect to imprecision.

1 INTRODUCTION

Dependent types support the development of extremely reliable software. With the full power of higher-order logic, programmers can write expressive specifications as types, and be confident that if a program typechecks, it meets its specification. Dependent types are at the core of proof assistants like Coq [Bertot and Castéran 2004] and Agda [Norell 2009]. While these can be used for certified programming [Chlipala 2013], their focus is on the construction of proofs, rather than generating practical or efficient code. Dependently-typed programming languages such as Idris [Brady 2013] aim to bring the benefits of dependent types to standard programming environments. Idris uses call-by-value semantics, with a clear phase distinction that prevents non-terminating code from running at compile-time. In this paper, we focus on dependently typed programming languages, not on proof assistants.

As with any static type system, the reliability brought by dependent types comes at a cost. Dependently-typed languages impose a rigid discipline, sometimes requiring programmers to explicitly construct proofs in their programs. Because of this rigidity, dependent types can interfere with rapid prototyping, and migrating code from languages with simpler type systems can be difficult. Several approaches have been proposed to relax dependent typing in order to ease programming, for instance by supporting some form of interoperability between simply-typed and dependently-typed programs and structures [Dagand et al. 2018; Osera et al. 2012; Ou et al. 2004;

Authors' addresses: Joseph Eremondi, Department of Computer Science, University of British Columbia, Canada, jeremond@cs.ubc.ca; Ronald Garcia, Department of Computer Science, University of British Columbia, Canada, jeremond@cs.ubc.ca; Éric Tanter, Computer Science Department (DCC), University of Chile, Chile, etanter@dcc.uchile.cl.

Tanter and Tabareau 2015]. These approaches require programmers to explicitly trigger runtime checks through casts, liftings, or block boundaries. Such explicit interventions hamper evolution.

In contrast, consider the philosophy of gradual typing [Siek and Taha 2006], which exploits *type imprecision* to drive the interaction between static and dynamic checking in a smooth, continuous manner [Siek et al. 2015]. A gradual language introduces an unknown type **?**, and admits imprecise types such as **Nat** $\rightarrow$ **?**. The gradual type system optimistically handles imprecision, deferring to runtime checks where needed. Therefore, runtime checking is an *implicit* consequence of type imprecision, and is seamlessly adjusted as programmers evolve the declared types of components, be they modules, functions, or expressions. This paper extends gradual typing to provide a flexible incremental path to adopting dependent types.

Gradual typing has been adapted to many other type disciplines, including ownership types [Sergey and Clarke 2012], effects [Bañados Schwerter et al. 2016], refinement types [Lehmann and Tanter 2017], security types [Fennell and Thiemann 2013; Toro et al. 2018a], and session types [Igarashi et al. 2017]. But it has not yet reached dependent types. Even as the idea holds much promise, it also poses significant challenges. The greatest barrier to gradual dependent types is that a dependent type checker must evaluate some program terms as part of type checking, and gradual types complicate this in two ways. First, if a gradual language fully embeds an untyped language, then some programs will diverge: indeed, self application $(\lambda x :: ?. x\ x)$ is typeable in such a language. Second, gradual languages introduce the possibility of runtime type errors: applying the function $(\lambda x :: ?.\ x + 1)$ may fail, depending on whether its argument's type is consistent with **Nat**. So gradual dependent types must account for how non-termination and failure occur *during typechecking*.

A gradual dependently-typed language. This work presents GDTL, a gradual dependently-typed core language that supports the whole spectrum between an untyped functional language and a dependently-typed one. As such, GDTL adopts a unified term and type language, meaning that the unknown type **?** is also a valid term. This allows programs to specify types with imprecise indices, and replace proof terms with **?** (Section 2).

GDTL is a gradual version of the predicative fragment of the Calculus of Constructions with a cumulative universe hierarchy (CC_ω) (Section 3), similar to the core language of Idris [Brady 2013]. We gradualize this language following the Abstracting Gradual Typing (AGT) methodology [Garcia et al. 2016] (Section 4). Because GDTL is a *conservative extension* of this dependently-typed calculus, it is both strongly normalizing and logically consistent for fully static code. On the dynamic side, GDTL can fully embed the untyped lambda calculus. When writing purely untyped code, static type errors are never encountered. In between, GDTL satisfies the *gradual guarantees* of Siek et al. [2015], meaning that typing and evaluation are monotone with respect to type imprecision. The guarantee ensures that programmers can move code between imprecise and precise types in small, incremental steps, with the program type-checking and running at each step.

Like Idris, GDTL is a call-by-value language with a sharp two-phase distinction. The key technical insight on which GDTL is built is to use two distinct notions of evaluation: one for *normalization* during typechecking, and one for *execution* at runtime. Specifically, we present a novel *approximate* normalization technique that guarantees decidable typechecking (Section 5): applying a function of unknown type, which may trigger non-termination, normalizes to the *unknown value* **?**. Consequently, some values that would be distinct at runtime become indistinguishable as type indices. This approximation ensures that compile-time normalization (i.e. during typechecking) always terminates and never signals a dynamic error. At runtime, GDTL uses the standard runtime execution strategy of gradual languages, which does not approximate **?**-typed functions, but may fail due to dynamic type errors, and may diverge (Section 6). We prove that GDTL has decidable typechecking and satisfies all the expected properties of gradual languages [Siek et al. 2015]: type safety, conservative extension of the static language, embedding of the untyped language, and the gradual guarantees (Section 7). We then show how inductive types with eliminators can be added to GDTL without significant changes (Section 8). Section 9 discusses related work, and Section 10 discusses limitations and perspectives for future work.

Disclaimer. This work does not aim to develop a full-fledged dependent gradual type theory, a fascinating objective that would raise many metatheoretic challenges. Rather, it proposes a novel technique applicable to call-by-value dependently-typed functional programming languages like Idris. Being a core calculus, GDTL currently lacks several features expected of a practical dependently-typed language like Idris, as detailed in Section 10. Nevertheless, this work provides a foundation on which such a practical gradual dependently-typed language can be built.

2 GOALS AND CHALLENGES

We begin by motivating our goals for GDTL, and describe the challenges and design choices that accompany them.

2.1 The Pain and Promise of Dependent Types

To introduce dependent types, we start with a classic example: length-indexed vectors. In a dependently-typed language, the type $\mathbf{Vec}\ \mathsf{A}\ \mathsf{n}$ describes any vector that contains n elements of type A. We say that **Vec** is *indexed* by the value n. This type has two constructors, $\mathbf{Nil} : (\mathsf{A} : \mathbf{Set}_1) \to \mathbf{Vec}\ \mathsf{A}\ 0$ and $\mathbf{Cons} : (\mathsf{A} : \mathbf{Set}_1) \to (\mathsf{n} : \mathbf{Nat}) \to \mathsf{A} \to \mathbf{Vec}\ \mathsf{A}\ \mathsf{n} \to \mathbf{Vec}\ \mathsf{A}\ (\mathsf{n} + 1)$. By making length part of the type, we ensure that operations that are typically partial can only receive values for which they produce results. For instance, one can type a function that yields the first element of a vector as follows:

$$\mathbf{head}\ :\ (\mathsf{A} : \mathbf{Set}_1) \to (\mathsf{n} : \mathbf{Nat}) \to \mathbf{Vec}\ \mathsf{A}\ (\mathsf{n} + 1) \to \mathsf{A}$$

Since head takes a vector of non-zero length, we can be sure that the function never receives an empty vector.

The downside of this strong guarantee is that we can only use vectors in contexts where their length is known. This makes it difficult to migrate code from languages with weaker types, or for newcomers to prototype algorithms. For example, a programmer may wish to migrate the following quicksort algorithm into a dependently-typed language:

```
sort vec =   if vec = Nil then Nil
                else (sort (filter (≤ (head vec)) (tail vec) ))++head vec
                  ++(sort (filter (> (head vec)) (tail vec) )))
```

Migrating this definition to a dependently-typed language poses some difficulties. The recursive calls are not direct deconstructions of vec, so it takes work to convince the type system that the code will terminate, and is thus safe to run at compile time. Moreover, if we try to use this definition with **Vec**, we must account for how the length of each filtered list is unknown, and while we can prove that the length of the resulting list is the same as the input, this must be done manually. Alternately, we could use simply-typed lists in the dependently-typed language, but we do not wish to duplicate every vector function for lists.

2.2 Gradual Types to the Rescue?

At first glance, gradual typing seems like it can provide the desired flexibility. In a gradually typed language, a programmer can use the unknown type, written **?**, to soften the static typing discipline. Terms with type **?** can appear in any context. Runtime type checks ensure that dynamically-typed code does not violate invariants expressed with static types.

Since **?** allows us to embed untyped code in a typed language, we can write a gradually-typed fixed-point combinator $\mathsf{Z} : (\mathsf{A} : \mathbf{Set}_1) \to (\mathsf{B} : \mathbf{Set}_1) \to ((\mathsf{A} \to \mathsf{B}) \to \mathsf{A} \to \mathsf{B}) \to \mathsf{A} \to \mathsf{B}$. This allows the programmer to write `sort` using general recursion. Furthermore, the programmer can give `sort` the type **?** → **?**, causing the length of the results of `filter` to be ignored in typechecking. Annotating the vector with **?** inserts runtime checks that ensure that the program will gracefully fail if it is given an argument that is not a vector.

However, introducing the dynamic type **?** in a dependently typed language brings unique challenges:

The Unknown Type is Not Enough. If we assign `sort` the type **?** → **?**, then we can pass it any argument, whether it is a vector or not. This seems like overkill: we want to restrict `sort` to vectors and statically rule out nonsensical calls like `sort false`. Unfortunately the usual notion of type imprecision is too coarse grained to support this. We want to introduce imprecision more judiciously, as in **Vec** A **?**: the type of vectors with *unknown length*. But the length is a natural number, not a type. How can we express imprecision in type indices?

Dependent Types Require Proofs. To seamlessly blend untyped and dependently-typed code to interact, we want to let programs omit proof terms, yet still allow code to typecheck and run. But this goes farther than imprecision in type indices, since imprecision also manifests in program terms. What should the dynamic semantics of imprecise programs be?

Gradual Typing Introduces Effects. Adding **?** to types introduces two effects. The ability to type self-applications means programs may diverge, and the ability to write imprecise types admits runtime type errors. These effects are troubling because dependent type checking must often evaluate code, sometimes under binders, to compare dependent types. For instance, the type checker should realize that $\lambda x.\, 1 + 1$ and $\lambda x.\, 2$ are the same thing. We must normalize terms at compile time to compute the type of dependent function applications. This means that both of effects can manifest *during type checking*. How should compile-time evaluation errors be handled? Can we make typechecking decidable in the presence of possible non-termination?

2.3 GDTL in Action

To propagate imprecision to type indices, and soundly omit proof terms, GDTL admits **?** *both as a type and a term.* To manage effects due to gradual typing, we use separate notions of evaluation for compile-time and runtime. Introducing *imprecision in the compile-time normalization* of types to avoids both non-termination and failures during typechecking.

The Unknown as a Type Index. Since dependently typed languages conflate types and terms, GDTL allows **?** to be used as either a term or a type. Just as any term can have type **?**, the term **?** can have any type. This lets dependent type checks be deferred to runtime. For example, we can define vectors `staticNil`, `dynNil` and `dynCons` as follows:

`staticNil` :	**Vec Nat** 0	`dynNil` :	**Vec Nat ?**	`dynCons` :	**Vec Nat ?**
`staticNil` =	**Nil Nat**	`dynNil` =	**Nil Nat**	`dynCons` =	**Cons Nat** 0 0 (**Nil Nat**)

Then, (`head` **Nat** 1 `staticNil`) does not typecheck, (`head` **Nat** 1 `dynNil`) typechecks but fails at runtime, and (`head` **Nat** 1 `dynCons`) typechecks and succeeds at runtime. The programmer can choose between compile-time or runtime-checks, but soundness is maintained either way, and in the fully-static case, the unsafe code is still rejected.

The Unknown as a Term at Runtime. Having **?** as a term means that programmers can use it to optimistically omit *proof terms*. Indeed, terms can be used not only as type indices, but also as proofs of invariants. For example, consider the type of equality proofs $\mathbf{Eq} : (\mathsf{A} : \mathbf{Set}_i) \to \mathsf{A} \to \mathsf{A} \to \mathbf{Set}_i$, along with its lone constructor $\mathbf{Refl} : (\mathsf{A} : \mathbf{Set}_i) \to (x : \mathsf{A}) \to \mathbf{Eq}\ \mathsf{A}\ x\ x$. We can use these to write a (slightly contrived) formulation of the head function:

$$\mathsf{head}' \ : \ (\mathsf{A} : \mathbf{Set}_i) \to (\mathsf{n} : \mathbf{Nat}) \to (\mathsf{m} : \mathbf{Nat}) \to \mathbf{Eq}\ \mathbf{Nat}\ \mathsf{n}\ (\mathsf{m} + 1) \to \mathbf{Vec}\ \mathsf{A}\ \mathsf{n} \to \mathsf{A}$$

This variant accepts vectors of any length, provided the user also supplies a proof that its length n is not zero (by providing the predecessor m and the equality proof $\mathbf{Eq}\ \mathsf{n}\ (\mathsf{m} + 1)$). GDTL allows **?** to be used in place of a proof, while still ensuring that a runtime error is thrown if head' is ever given an empty list. For instance, suppose we define a singleton vector and a proof that $0 = 0$:

$$\begin{array}{llll} \mathtt{staticCons} : & \mathbf{Vec}\ \mathbf{Nat}\ 1 & \mathtt{staticProof} : & \mathbf{Eq}\ \mathbf{Nat}\ 0\ 0 \\ \mathtt{staticCons} = & \mathbf{Cons}\ \mathbf{Nat}\ 0\ (\mathbf{Nil}\ \mathbf{Nat}) & \mathtt{staticProof} = & \mathbf{Refl}\ \mathbf{Nat}\ 0 \end{array}$$

Then (head' **Nat** 0 0 `staticProof` `staticNil`) does not typecheck, (head' **Nat** 0 **? ?** `staticNil`) typechecks but fails at runtime, and (head' **Nat** 1 **? ?** `staticCons`) typechecks and succeeds at runtime. To see why we get a runtime failure for `staticNil`, we note that internally, head' uses a *rewriting* property of equality, which says that any property P that holds for x must hold for y if they are equal:

$$\mathtt{rewrite} \ : \ (\mathsf{A} : \mathbf{Set}_1) \to (x : \mathsf{A}) \to (y : \mathsf{A}) \to (\mathsf{P} : \mathsf{A} \to \mathbf{Set}_1) \to \mathbf{Eq}\ \mathsf{A}\ x\ y \to \mathsf{P}\ x \to \mathsf{P}\ y$$

Following the intuition of AGT [Garcia et al. 2016], the unknown type **?** stands in for all plausible static types. Similarly, *the gradual term ? denotes all possible static terms*. Thus applying the term **?** as a function represents applying all possible functions, producing all possible results, and summarizing those results as a gradual term **?** again. This means that **?**, when applied as a function, behaves as $\lambda x.\,?$. Similarly, **?** treated as an equality proof behaves as **Refl ? ?**.

Because **?** behaves as **Refl ? ?** in GDTL, in the latter two cases of our example, `rewrite` gives a result of type P **?**, which is checked against type $\mathsf{P}\ y$ *at runtime*. If $\mathsf{P}\ x$ and $\mathsf{P}\ y$ are not the same at runtime, then this check fails, causing a runtime error.

Managing Effects from Gradual Typing. For an example of how the effects of gradual typing can show up in typechecking, we use the aforementioned Z combinator to accidentally write a non-terminating function `badFact`.

$$\mathtt{badFact} = \lambda\, m\, .\, Z\ (\lambda\, f\ .\, \mathtt{ifzero}\ m\ (f\ 1)(m * f\ (m))\qquad \text{-- } \textit{never terminates}$$

As explained before, from a practical point of view, it is desirable for GDTL to fully support dynamically-typed terms, because it allows the programmer to opt out of both the type discipline and the termination discipline of a dependently-typed language. However, this means that computing the return type of a function application may diverge, for instance:

$$\begin{array}{lll} \mathsf{repeat} : & (\mathsf{A} : \mathbf{Set}_1) \to (\mathsf{n} : \mathbf{Nat}) \to \mathsf{A} \to (\mathbf{Vec}\ \mathsf{A}\ \mathsf{n}) & \\ \mathsf{factList} = & \mathtt{repeat}\ \mathbf{Nat}\ (\mathtt{badFact}\ 1)\ 0 & \text{-- } \textit{has type } \mathbf{Vec}\ \mathsf{A}\ (\mathtt{badFact}\ \mathit{1}) \end{array}$$

To isolate the non-termination from imprecise code, we observe that any diverging code will necessarily apply a function of type **?**.

Similarly, a naive approach to gradual dependent types will encounter failures when normalizing some terms. Returning to our head function, how should we typecheck the following term?

$$\mathtt{failList} = \mathsf{head}\ \mathbf{Nat}\ (\mathtt{false} :: ?)\ \mathtt{staticCons}$$

We need to check `staticCons` against **Vec Nat** ((`false` :: **?**) + 1), but what meaning does (`false` :: **?**) + 1 have as the length of a vector?

The difficulty here is, if a term contains type ascriptions that may produce a runtime failure, then it will *always* trigger an error when normalizing it, since we evaluate under binders and remove ascriptions to produce a normal form. This means that typechecking will fail any time we apply a function to a possibly-failing term. This is highly undesirable, and goes against the spirit of gradual typing: writing programs in the large would be very difficult if applying a function to an argument that does match its domain type caused a type error, or caused typechecking to diverge!

GDTL avoids both problems by using different notions of running programs for the compile-time and runtime phases. We distinguish compile-time *normalization*, which is approximate but total, from runtime *execution*, which is exact but may diverge. When non-termination or failures are possible, compile-time normalization uses **?** as an approximate but pure result. So both `factList` and `failList` can be defined and used in runtime code, but they are assigned type **Vec Nat ?**. To avoid non-termination and dynamic failures, we want our language to be strongly normalizing during type-checking. Approximate normalization gives us this.

Our normalization is focused around *hereditary substitution* [Watkins et al. 2003], which is a total operation from canonical forms to canonical forms. Because hereditary substitution is structurally decreasing in the *type* of the value being substituted, a static termination proof is easily adapted to GDTL.

This allows us to pinpoint exactly where gradual types introduces effects, approximate in those cases, and easily adapt the proof of termination of a static language to the gradual language GDTL. Similarly, our use of bidirectional typing means that a single check needs to be added to prevent failures in normalization.

2.4 The Gradual Guarantee

To ensure a smooth transition between precise and imprecise typing, GDTL satisfies the *gradual guarantee*, which comes in two parts [Siek et al. 2015]. The *static gradual guarantee* says that reducing the precision of a type ascription must always preserve well-typedness. If the programmer decides a piece of code is too precisely typed, they can relax its precision, knowing that this will not cause a static type error. The *dynamic gradual guarantee* that reducing the precision of a type ascription preserves program behaviour, though the resulting value may be less precise.

One novel insight of GDTL's design is that the interplay between dependent type checking and program evaluation carries over to the gradual guarantees. Specifically, the static gradual guarantee fundamentally depends on a restricted variant of the dynamic gradual guarantee. We show that approximate normalization maps terms related by precision to canonical forms related by precision, thereby ensuring that reducing a term's precision always preserves well-typedness.

By satisfying the gradual typing criteria, combined with our embedding properties, GDTL gives programmers freedom to move within the entire spectrum of typedness, from the safety of higher-order logic to the flexibility of the dynamic languages. What's more, admitting **?** as a term means that we can easily combine dependently typed code with simply typed code, the midpoint between dynamic and dependent types. For example, the simple list type could be written as $\mathbf{List}\ \mathsf{A} = \mathbf{Vec}\ \mathsf{A}\ \mathbf{?}$, so lists could be given to vector-expecting code and vice-versa. The programmer knows that as long as vectors are used in vector-expecting code, no crashes can happen, and soundness ensures that using a list in a vector operation will always fail gracefully or run successfully. This is significantly different from existing gradual dependent type work [Dagand et al. 2018; Tanter and Tabareau 2015], where the user must provide explicit equivalences or decidable properties.

3 SDTL: A STATIC DEPENDENTLY-TYPED LANGUAGE

We now present SDTL, a static dependently-typed language which is essentially a bidirectional, call-by-value, cumulative variant of the predicative fragment of CC_ω(i.e. the calculus of constructions

$$
\begin{array}{lll}
\textbf{Static Terms} & \textbf{Simple Values} & \textbf{Simple Contexts} \\
\begin{array}{rcl} \mathsf{t}, \mathsf{T} & ::= & \\ & \mid & \lambda x.\,\mathsf{t} \\ & \mid & \mathsf{t}_1\ \mathsf{t}_2 \\ & \mid & x \\ & \mid & (x : \mathsf{T}_1) \to \mathsf{T}_2 \\ & \mid & \mathbf{Set}_i \\ & \mid & \mathsf{t} :: \mathsf{T} \end{array}
&
\begin{array}{rcl} \mathsf{v}, \mathsf{V} & ::= & \\ & \mid & \lambda x.\,\mathsf{t} \\ & \mid & (x : \mathsf{V}) \to \mathsf{T} \\ & \mid & \mathbf{Set}_i \end{array}
&
\begin{array}{rcl} C & ::= & \\ & \mid & \square\,\mathsf{t} \\ & \mid & \mathsf{v}\,\square \\ & \mid & (x : \square) \to \mathsf{T} \\ & \mid & \square :: \mathsf{T} \end{array}
\end{array}
$$

$\boxed{\mathsf{t}_1 \longrightarrow \mathsf{t}_2}$ ***(Simple Small-Step Semantics)***

$$
\textsc{SimpleStepAnn}\ \frac{}{(\mathsf{v} :: \mathsf{T}) \longrightarrow \mathsf{v}}
\qquad
\textsc{SimpleStepApp}\ \frac{}{(\lambda x.\,\mathsf{t})\ \mathsf{v} \longrightarrow [x \Mapsto \mathsf{v}]\mathsf{t}}
\qquad
\textsc{SimpleStepContext}\ \frac{\mathsf{t}_1 \longrightarrow \mathsf{t}_2}{C[\mathsf{t}_1] \longrightarrow C[\mathsf{t}_2]}
$$

Fig. 1. SDTL: Syntax and Semantics

with a universe hierarchy [Coquand and Huet 1988]). SDTL is the starting point of our gradualization effort, following the Abstracting Gradual Typing (AGT) methodology [Garcia et al. 2016], refined to accommodate dependent types.

3.1 Syntax and Dynamic Semantics

The syntax of SDTL is shown in Figure 1. Metavariables in SDTL are written in red, sans-serif font. Types and terms share a syntactic category. Functions and applications are in their usual form. Function types are *dependent*: a variable name is given to the argument, and the codomain may refer to this variable. We have a *universe hierarchy*: the lowest types have the type $\mathbf{Set}_1$, and each $\mathbf{Set}_i$ has type $\mathbf{Set}_{i+1}$. This hierarchy is *cumulative*: any value in $\mathbf{Set}_i$ is also in $\mathbf{Set}_{i+1}$. Finally, we have a form for explicit type ascriptions.

We use metavariables v, V to range over static values, which are the subset of terms consisting only of functions, function types and universes. For evaluation, we use a call-by-value reduction semantics (Figure 1). Ascriptions are dropped when evaluating, and function applications result in (syntactic) substitution. We refer to the semantics as *simple* rather than *static*, since they apply equally well to an untyped calculus, albeit without the same soundness guarantees.

3.2 Comparing Types: Canonical Forms

Since dependent types can contain expressions, it is possible that types may contain redexes. Most dependent type systems have a *conversion* rule that assigns an expression type T_1 if it has type T_2, and T_2 is convertible into T_1 through some sequence of β-conversions, η-contractions and expansions, and α-renamings. Instead, we treat types as $\alpha\beta\eta$-equivalence classes. To compare equivalence classes, we represent them using *canonical forms* [Watkins et al. 2003]. These are β-reduced, η-long canonical members of an equivalence class. We compare terms for $\alpha\beta\eta$-equivalence by normalizing and syntactically comparing their canonical forms.

The syntax for canonical forms is given in Figure 2. We omit well-formedness rules for terms and environments, since the only difference from the typing rules is the η-longness check.

By representing function applications in *spine form* [Cervesato and Pfenning 2003], we can ensure that all heads are variables, and thus no redexes are present, even under binders. The well-formedness of canonical terms is ensured using bidirectional typing [Pierce and Turner 2000]. A variable x or a universe $\mathbf{Set}_i$ is an *atomic form*. Our well-formedness rules ensure the types of

$\mathsf{u}, \mathsf{U} \in \text{SCanonical}$

Static Canonical Forms

$$\begin{array}{lll} \mathsf{u},\ \mathsf{U} & ::= & \\ & \mid & \lambda x.\, \mathsf{u} \\ & \mid & \mathsf{r} \\ & \mid & (x : \mathsf{U}_1) \to \mathsf{U}_2 \end{array}$$

Static Atomic Forms

$$\begin{array}{lll} \mathsf{r},\ \mathsf{R} & ::= & \\ & \mid & x\bar{\mathsf{s}} \\ & \mid & \mathbf{Set}_i \end{array}$$

Static Canonical Spines

$$\begin{array}{lll} \bar{\mathsf{s}} & ::= & \\ & \mid & \cdot \\ & \mid & \bar{\mathsf{s}}\ \mathsf{u} \end{array}$$

$\boxed{\Gamma \vdash \mathsf{t} \Rightarrow \mathsf{U} \quad \mid \quad \Gamma \vdash \mathsf{t} \Leftarrow \mathsf{U}}$ ***(Static Typing: Synthesis and Checking)***

$$\textsc{SSynthAnn}\ \frac{\Gamma \vdash \mathsf{U} \leftsquigarrow \mathsf{T} : \mathbf{Set} \qquad \Gamma \vdash \mathsf{t} \Leftarrow \mathsf{U}}{\Gamma \vdash (\mathsf{t} :: \mathsf{T}) \Rightarrow \mathsf{U}}
\qquad
\textsc{SSynthApp}\ \frac{\Gamma \vdash \mathsf{t}_1 \Rightarrow (x : \mathsf{U}_1) \to \mathsf{U}_2 \qquad \Gamma \vdash \mathsf{u} \leftsquigarrow \mathsf{t}_2 \Leftarrow \mathsf{U}_1 \qquad [\mathsf{u}/x]^{\mathsf{U}_1}\mathsf{U}_2 = \mathsf{U}_3}{\Gamma \vdash \mathsf{t}_1\ \mathsf{t}_2 \Rightarrow \mathsf{U}_3}$$

$$\textsc{SSynthVar}\ \frac{\vdash \Gamma \qquad (x : \mathsf{U}) \in \Gamma}{\Gamma \vdash x \Rightarrow \mathsf{U}}
\qquad
\textsc{SSynthSet}\ \frac{i > 0}{\Gamma \vdash \mathbf{Set}_i \Rightarrow \mathbf{Set}_{i+1}}
\qquad
\textsc{SCheckSynth}\ \frac{\Gamma \vdash \mathsf{t} \Rightarrow \mathsf{U}}{\Gamma \vdash \mathsf{t} \Leftarrow \mathsf{U}}
\qquad
\textsc{SCheckLevel}\ \frac{\Gamma \vdash \mathsf{T} \Rightarrow \mathbf{Set}_i \qquad 0 < i < j}{\Gamma \vdash \mathsf{T} \Leftarrow \mathbf{Set}_j}$$

$$\textsc{SCheckLam}\ \frac{\begin{array}{c} \vdash (x : \mathsf{U}_1)\Gamma \\ (x : \mathsf{U}_1)\Gamma \vdash \mathsf{t} \Leftarrow \mathsf{U}_2 \end{array}}{\Gamma \vdash (\lambda x.\, \mathsf{t}) \Leftarrow (x : \mathsf{U}_1) \to \mathsf{U}_2}
\qquad
\textsc{SCheckPi}\ \frac{\Gamma \vdash \mathsf{U} \leftsquigarrow \mathsf{T}_1 \Leftarrow \mathbf{Set}_i \qquad \vdash (x : \mathsf{U})\Gamma \qquad (x : \mathsf{U})\Gamma \vdash \mathsf{T}_2 \Leftarrow \mathbf{Set}_i}{\Gamma \vdash (x : \mathsf{T}_1) \to \mathsf{T}_2 \Leftarrow \mathbf{Set}_i}$$

Fig. 2. SDTL: Canonical Forms and Typing Rules

atomic forms are themselves atomic. This ensures that canonical forms are η-long, since no variable has type $(y : \mathsf{U}_1) \to \mathsf{U}_2$.

3.3 Type-Checking and Normalization

Using the concept of canonical forms, we can now express the type rules for SDTL in Figure 2. To ensure syntax-directedness, we again use bidirectional typing.

The *type synthesis* judgement $\Gamma \vdash \mathsf{t} \Rightarrow \mathsf{U}$ says that t has type U under context Γ, where the type is treated as an output of the judgement. That is, from examining the term, we can determine its type. Conversely, the *checking* judgment $\Gamma \vdash \mathsf{t} \Leftarrow \mathsf{U}$ says that, given a type U, we can confirm that t has that type. These rules allow us to propagate the information from ascriptions inwards, so that only top-level terms and redexes need ascriptions.

Most rules in the system are standard. To support dependent types, SSynthApp computes the result of applying a particular value. We switch between checking and synthesis using SSynthAnn and SCheckSynth. The predicativity of our system is distilled in the SSynthSet rule: $\mathbf{Set}_i$ always has type $\mathbf{Set}_{i+1}$. The rule SCheckLevel encodes *cumulativity*: we can always treat types as if they were in a higher level universe, though the converse does not hold. Cumulativity allows us to check function types against any $\mathbf{Set}_i$ in SCheckPi, provided the domain and codomain both check against that $\mathbf{Set}_i$.

We distinguish *hereditary substitution* on canonical forms $[\mathsf{u}_1/x]^{\mathsf{U}}\mathsf{u}_2 = \mathsf{u}_3$, from *syntactic substitution* $[x \Mapsto \mathsf{t}_1]\mathsf{t}_2 = \mathsf{t}_3$ on terms. Notably, the former takes the type of its variable as input, and has canonical forms as both inputs and as output. In SSynthApp and SCheckPi, we use the *normalization* judgement $\Gamma \vdash \mathsf{u} \leftsquigarrow \mathsf{t} \Leftarrow \mathsf{U}$, which computes the canonical form of t while checking

$\boxed{\Gamma \vdash \mathsf{U} \leftsquigarrow \mathsf{T} : \mathbf{Set}}$ ***(Set Normalization (rules omitted))***

$\boxed{\Gamma \vdash \mathsf{t} \rightsquigarrow \mathsf{u} \Rightarrow \mathsf{U} \quad | \quad \Gamma \vdash \mathsf{u} \leftsquigarrow y \Leftarrow \mathsf{U}}$ ***(Static Normalization)***

$$\textsc{SNormSynthAnn}\ \frac{\begin{array}{c}\Gamma \vdash \mathsf{U} \leftsquigarrow \mathsf{T} : \mathbf{Set} \\ \Gamma \vdash \mathsf{u} \leftsquigarrow \mathsf{t} \Leftarrow \mathsf{U}\end{array}}{\Gamma \vdash (\mathsf{t} :: \mathsf{T}) \rightsquigarrow \mathsf{u} \Rightarrow \mathsf{U}} \qquad \textsc{SNormSynthVar}\ \frac{\begin{array}{c}\vdash \Gamma \\ (x : \mathsf{U}) \in \Gamma \quad x \rightsquigarrow_\eta \mathsf{u} : \mathsf{U}\end{array}}{\Gamma \vdash x \rightsquigarrow \mathsf{u} \Rightarrow \mathsf{U}}$$

$$\textsc{SNormSynthApp}\ \frac{\begin{array}{c}\Gamma \vdash \mathsf{t}_1 \rightsquigarrow (\lambda x.\, \mathsf{u}_1) \Rightarrow (x : \mathsf{U}_1) \to \mathsf{U}_2 \\ \Gamma \vdash \mathsf{u}_2 \leftsquigarrow \mathsf{t}_2 \Leftarrow \mathsf{U}_1 \\ [\mathsf{u}_2/x]^{\mathsf{U}_1}\mathsf{u}_1 = \mathsf{u}_3 \qquad [\mathsf{u}_2/x]^{\mathsf{U}_1}\mathsf{U}_2 = \mathsf{U}_3\end{array}}{\Gamma \vdash \mathsf{t}_1\ \mathsf{t}_2 \rightsquigarrow \mathsf{u}_3 \Rightarrow \mathsf{U}_3}$$

$\boxed{[\mathsf{u}_1/x]^{\mathsf{U}}\mathsf{u}_2 = \mathsf{u}_3}$ ***(Static Hereditary Substitution)***

$$\textsc{SHsubPi}\ \frac{\begin{array}{c}[\mathsf{u}/x]^{\mathsf{U}}\mathsf{U}_1 = \mathsf{U}_1' \\ [\mathsf{u}/x]^{\mathsf{U}}\mathsf{U}_2 = \mathsf{U}_2' \quad x \neq y\end{array}}{[\mathsf{u}/x]^{\mathsf{U}}(y : \mathsf{U}_1) \to \mathsf{U}_2 = (y : \mathsf{U}_1') \to \mathsf{U}_2'} \qquad \textsc{SHsubDiffNil}\ \frac{x \neq y}{[\mathsf{u}/x]^{\mathsf{U}}\, y = y} \qquad \textsc{SHsubDiffCons}\ \frac{\begin{array}{c}x \neq y \quad [\mathsf{u}/x]^{\mathsf{U}}\, y\bar{\mathsf{s}} = y\bar{\mathsf{s}}' \\ [\mathsf{u}/x]^{\mathsf{U}}\mathsf{u}_2 = \mathsf{u}_3\end{array}}{[\mathsf{u}/x]^{\mathsf{U}}\, y\bar{\mathsf{s}}\ \mathsf{u}_2 = y\bar{\mathsf{s}}'\ \mathsf{u}_3}$$

$$\textsc{SHsubSet}\ \frac{}{[\mathsf{u}/x]^{\mathsf{U}}\mathbf{Set}_i = \mathbf{Set}_i} \qquad \textsc{SHsubLam}\ \frac{[\mathsf{u}/x]^{\mathsf{U}}\mathsf{u}_2 = \mathsf{u}_3 \quad x \neq y}{[\mathsf{u}/x]^{\mathsf{U}}(\lambda y.\, \mathsf{u}_2) = (\lambda y.\, \mathsf{u}_3)} \qquad \textsc{SHsubSpine}\ \frac{[\mathsf{u}_1/x]^{\mathsf{U}}\, x\bar{\mathsf{s}}\ \mathsf{u}_2 = \ \mathsf{u}_3 \ : \ \mathsf{U}'}{[\mathsf{u}_1/x]^{\mathsf{U}}\, x\bar{\mathsf{s}}\ \mathsf{u}_2 = \mathsf{u}_3}$$

$\boxed{[\mathsf{u}/x]^{\mathsf{U}}\, x\bar{\mathsf{s}} = \ \mathsf{u}' \ : \ \mathsf{U}'}$ ***(Static Atomic Hereditary Substitution)***

$$\textsc{SHsubRHead}\ \frac{}{[\mathsf{u}/x]^{\mathsf{U}}\, x = \ \mathsf{u} \ : \ \mathsf{U}} \qquad \textsc{SHsubRSpine}\ \frac{\begin{array}{c}[\mathsf{u}_1/x]^{\mathsf{U}}\, x\bar{\mathsf{s}} = \ (\lambda y.\, \mathsf{u}_1') \ : \ (y : \mathsf{U}_1') \to \mathsf{U}_2' \qquad [\mathsf{u}_1/x]^{\mathsf{U}}\mathsf{u}_2 = \mathsf{u}_3 \\ [\mathsf{u}_3/y]^{\mathsf{U}_1'}\mathsf{u}_1' = \mathsf{u}_2' \qquad [\mathsf{u}_3/y]^{\mathsf{U}_1'}\mathsf{U}_2' = \mathsf{U}_3'\end{array}}{[\mathsf{u}_1/x]^{\mathsf{U}}\, x\bar{\mathsf{s}}\ \mathsf{u}_2 = \ \mathsf{u}_2' \ : \ \mathsf{U}_3'}$$

Fig. 3. SDTL: Normalization (select rules) and Hereditary Substitution

it against U. Similarly, SSynthAnn uses the judgement $\Gamma \vdash \mathsf{U} \leftsquigarrow \mathsf{T} : \mathbf{Set}$, which computes the canonical form of T while ensuring it checks against some $\mathbf{Set}_i$.

The rules for normalization (Figure 3) directly mirror those for well-typed terms, building up the canonical forms from sub-derivations. In particular, the rule SNormSynthVar η-expands any variables with function types, which allows us to assume that the function in an application will always normalize to a λ-term. (The rules for the eta expansion function $x \rightsquigarrow_\eta \mathsf{u} : \mathsf{U}$ are standard, so we omit them). We utilize this assumption in SNormSynthApp, where the canonical form of an application is computed using hereditary substitution.

3.4 Hereditary Substitution

Hereditary substitution is defined in Figure 3. At first glance, many of the rules look like a traditional substitution definition. We traverse the expression looking for variables, and replace them with the corresponding term.

However, there are some key differences. Hereditary substitution has canonical forms as both inputs and outputs. The key work takes place in the rule SHsubRSpine. When replacing x with u_1 in $x\bar{\mathsf{s}}\ \mathsf{u}_2$, we find the substituted forms of u_2 and $x\bar{\mathsf{s}}$, which we call u_3 and $\lambda y.\, \mathsf{u}_1'$ respectively. If the inputs are well typed and η-long, the the substitution of the spine will always return a λ-term, meaning that its application to u_3 is not a canonical form. To produce a canonical form in such a case, we continue substituting, recursively replacing y with u_3 in u'_1 A similar substitution in the the codomain of U gives our result type. Thus, if this process terminates, it will always produce a canonical form.

To ensure that the process does, in fact, terminate for well-typed inputs, we define hereditary substitution in terms of the type of the variable being replaced. Since we are replacing a different variable in the premise StaticHsubRSpine, we must keep track of the type of the resultant expression when substituting in spines, which is why substitution on atomic forms is a separate relation. We order types by the multiset of universes of all arrow types that are subterms of the type, similar to techniques used for Predicative System F [Eades and Stump 2010; Mangin and Sozeau 2015]. We can use the well-founded multiset ordering given by Dershowitz and Manna [1979]: if a type U has maximum arrow type universe i, we say that it is greater than all other types containing fewer arrows at universe i whose maximum is not greater than i. Predicativity ensures that, relative to this ordering, the return type of a function application is always less than the type of the function itself. In all premises but the last two of SHsubRSpine, we recursively invoke substitution on strict-subterms, while keeping the type of the variable the same. In the remaining cases, we observe that we are performing substitution at a type that is smaller by our multiset order.

3.5 Properties of SDTL

Since SDTL is mostly standard, it enjoys the standard properties of dependently-typed languages. Hereditary substitution can be used to show that the language is strongly normalizing, and thus consistent as a logic. Since the type rules, hereditary substitution and normalization are syntax directed and terminating, typechecking is decidable. Finally, because all well-typed terms have canonical forms, SDTL is type sound.

4 GDTL: ABSTRACTING THE STATIC LANGUAGE

We now present GDTL, a gradual counterpart of SDTL derived following an extension of the Abstracting Gradual Typing (AGT) methodology [Garcia et al. 2016] to the setting of dependent types. We start by giving meaning to gradual types and terms as abstractions of sets of static entities, and use this meaning to derive the typing rules of GDTL.

Throughout this section, we assume that we already have gradual counterparts of both hereditary substitution and normalization. We leave the detailed development of these notions to Section 5, as they are are non-trivial if one wants to preserve both decidable type checking and the gradual guarantee (Section 2.4). The dynamic semantics of GDTL are presented in Section 6, and its metatheory in Section 7.

4.1 Terms and Canonical Forms

Syntax. The syntax of GDTL (Figure 4) is a simple extension of SDTL's syntax. Metavariables in GDTL are written in blue, serif font. The syntax additionally includes the unknown term/type, **?**. This represents a type or term that is unknown to the programmer: by annotating a type with **?** or leaving a term as **?**, they can allow their program to typecheck with only partial typing information or an incomplete proof. Similarly, **?** is added to the syntax of canonical values.

Canonical forms do not contain ascriptions. While static types use ascriptions only for guiding type checking, the potential for dynamic type failure means that ascriptions have computational content in gradual typing. Notably, only variables can synthesize **?** as a type, though any typed expression can be checked against **?**. This allows us to reason about canonical forms at a given

$$\mathrm{u}, \mathrm{U} \in \textsc{GCanonical}$$

Gradual Terms

$$\begin{array}{lcl} \mathrm{t},\ \mathrm{T} & ::= & \\ & \mid & \lambda x.\, \mathrm{t} \\ & \mid & \mathrm{t}_1\ \mathrm{t}_2 \\ & \mid & x \\ & \mid & \mathbf{Set}_i \\ & \mid & (x : \mathrm{T}_1) \longrightarrow \mathrm{T}_2 \\ & \mid & \mathrm{t} :: \mathrm{T} \\ & \mid & ? \end{array}$$

Gradual Canonical Forms

$$\begin{array}{lcl} \mathrm{u},\ \mathrm{U} & ::= & \\ & \mid & \lambda x.\, \mathrm{u} \\ & \mid & \mathrm{r} \\ & \mid & ? \\ & \mid & (x : \mathrm{U}_1) \longrightarrow \mathrm{U}_2 \end{array}$$

Gradual Atomic Forms

$$\begin{array}{lcl} \mathrm{r},\ \mathrm{R} & ::= & \\ & \mid & x\bar{\mathrm{s}} \\ & \mid & \mathbf{Set}_i \end{array}$$

Gradual Canonical Spines

$$\begin{array}{lcl} \bar{\mathrm{s}} & ::= & \\ & \mid & \cdot \\ & \mid & \bar{\mathrm{s}}\ \mathrm{u} \end{array}$$

Fig. 4. GDTL: Terms and Canonical Forms

$$\gamma : \textsc{GCanonical} \longrightarrow \mathcal{P}(\textsc{SCanonical}) \setminus \emptyset$$

$$\begin{array}{llll} \gamma(?) & = \textsc{SCanonical} & \gamma(\lambda x.\, \mathrm{u}) & = \{\, \lambda x.\, \mathrm{u}' \mid \mathrm{u}' \in \gamma(\mathrm{u}) \,\} \\ \gamma(\mathbf{Set}_i) & = \{\, \mathbf{Set}_i \,\} & \gamma(x\bar{\mathrm{s}}\ \mathrm{u}) & = \{\, x\bar{\mathrm{s}}'\ \mathrm{u}' \mid x\bar{\mathrm{s}}' \in \gamma(x\bar{\mathrm{s}}), \mathrm{u}' \in \gamma(\mathrm{u}) \,\} \\ \gamma(x) & = \{\, x \,\} & \gamma((x : \mathrm{U}_1) \longrightarrow \mathrm{U}_2) & = \{\, (x : \mathrm{U}_1') \longrightarrow \mathrm{U}_2' \mid \mathrm{U}_1' \in \gamma(\mathrm{U}_1), U'2 \in \gamma(\mathrm{U}_2) \,\} \end{array}$$

$\boxed{\mathrm{U} \cong \mathrm{U}'}$ ***(Consistency of Gradual Canonical Terms)***

$$\textsc{ConsistentEq}\ \frac{}{\mathrm{u} \cong \mathrm{u}} \qquad \textsc{ConsistentPi}\ \frac{\mathrm{U}_1 \cong \mathrm{U}_1' \quad \mathrm{U}_2 \cong \mathrm{U}_2'}{(x : \mathrm{U}_1) \longrightarrow \mathrm{U}_2 \cong (x : \mathrm{U}_1') \longrightarrow \mathrm{U}_2'} \qquad \textsc{ConsistentLam}\ \frac{\mathrm{u} \cong \mathrm{u}'}{(\lambda x.\, \mathrm{u}) \cong (\lambda x.\, \mathrm{u}')}$$

$$\textsc{ConsistentApp}\ \frac{x\bar{\mathrm{s}} \cong x\bar{\mathrm{s}}' \quad \mathrm{u} \cong \mathrm{u}'}{x\bar{\mathrm{s}}\ \mathrm{u} \cong x\bar{\mathrm{s}}'\ \mathrm{u}'} \qquad \textsc{ConsistentDynL}\ \frac{}{? \cong \mathrm{u}} \qquad \textsc{ConsistentDynR}\ \frac{}{\mathrm{u} \cong ?}$$

Fig. 5. GDTL: Concretization and Consistency

type: while we can layer ascriptions on terms, such as $(\mathtt{true} :: ?) :: ? \longrightarrow ?$, the only canonical forms with function types are lambdas and $?$.

4.2 Concretization and Predicates

The main idea of AGT is that gradual types abstract sets of static types, and that each gradual type can be made *concrete* as a set of static types. For our system, we simply extend this to say that gradual terms represent sets of static terms. In a simply typed language, a static type embedded in the gradual language concretizes to the singleton set containing itself. However, for terms, we wish to consider the entire $\alpha\beta\eta$-equivalence class of the static term. As with typechecking, this process is facilitated by considering only canonical forms. The concretization function $\gamma : \textsc{GCanonical} \longrightarrow \mathcal{P}(\textsc{SCanonical})$, defined in Figure 5, recurs over sub-terms, with $?$ mapping to the set of all terms.

Given the concretization, we can *lift* a predicate from the static system to the gradual system. A predicate holds for gradual types if it holds for some types in their concretizations. For equality, this means that $\mathrm{U} \cong \mathrm{U}'$ if and only if $\gamma(\mathrm{U}) \cap \gamma(\mathrm{U}') \neq \emptyset$. We present a syntactic version of this in Figure 5. Concretization gives us a notion of *precision* on gradual types. We say that $\mathrm{U} \sqsubseteq \mathrm{U}'$ if $\gamma(\mathrm{U}) \subseteq \gamma(\mathrm{U}')$: that is, U is more precise because there are fewer terms it could plausibly represent.

$$\alpha : \mathcal{P}(\textsc{SCanonical}) \setminus \emptyset \to \textsc{GCanonical}$$

$$\begin{array}{rll}
\alpha(\{\, x\bar{s}\ \mathsf{u} \mid x\bar{s} \in A, \mathsf{u} \in B \,\}) = & x\bar{s}'\ \mathsf{u}' & \text{where } \alpha(A) = x\bar{s}', \alpha(B) = \mathsf{u}' \\
\alpha(\{\, (x : \mathsf{U}_1) \to \mathsf{U}_2 \mid \mathsf{U}_1 \in A, \mathsf{U}_2 \in B \,\}) = & (x : \mathsf{U}_1') \to \mathsf{U}_2' & \text{where } \alpha(A) = \mathsf{U}_1', \alpha(B) = \mathsf{U}_2' \\
\alpha(\{\, \lambda x.\, \mathsf{u} \mid \mathsf{u} \in A \,\}) = & \lambda x.\, \mathsf{u}' & \text{where } \alpha(A) = \mathsf{u}'
\end{array}$$

$$\alpha(\{\, \mathbf{Set}_i \,\}) \ = \ \mathbf{Set}_i \qquad \alpha(\{\, x \,\}) \ = \ x \qquad \alpha(A) \ = \ ?\ \text{otherwise}$$

$$\boxed{\mathbf{dom} : \textsc{GCanonical} \rightharpoonup \textsc{GCanonical}} \qquad \boxed{[\square/\_]^{\square}\ \mathbf{cod}\square : \textsc{GCanonical}^3 \rightharpoonup \textsc{GCanonical}}$$

$$\boxed{[\square/\_]^{\square}\ \mathbf{body}\square : \textsc{GCanonical}^3 \rightharpoonup \textsc{GCanonical}} \qquad \textit{(Partial Functions)}$$

$$\textsc{DomainPi}\ \frac{}{\mathbf{dom}\ (x : \mathsf{U}_1) \to \mathsf{U}_2 = \mathsf{U}_1} \qquad \textsc{CodSubPi}\ \frac{[\mathsf{u}/x]^{\mathsf{U}_1}\mathsf{U}_2 = \mathsf{U}_2'}{[\mathsf{u}/\_]\mathbf{cod}\ (x : \mathsf{U}_1) \to \mathsf{U}_2 = \mathsf{U}_2'} \qquad \textsc{BodySubPi}\ \frac{[\mathsf{u}/x]^{\mathsf{U}}\mathsf{u}_2 = \mathsf{u}_2'}{[\mathsf{u}/\_]^{\mathsf{U}}\mathbf{body}\ (\lambda x.\, \mathsf{u}_2) = \mathsf{u}_2'}$$

$$\textsc{DomainDyn}\ \frac{}{\mathbf{dom}\ ? = ?} \qquad \textsc{CodSubDyn}\ \frac{}{[\mathsf{u}/\_]\mathbf{cod}\ ? = ?} \qquad \textsc{BodySubDyn}\ \frac{}{[\mathsf{u}/\_]^{\mathsf{U}}\mathbf{body}\ ? = ?}$$

Fig. 6. GDTL: Abstraction and Lifted Functions

We can similarly define $\mathsf{U} \sqcap \mathsf{U}'$ as the most general term that is as precise as both U and U'. Note that $\mathsf{U} \sqcap \mathsf{U}'$ is defined if and only if $\mathsf{U} \cong \mathsf{U}'$ i.e. if $\gamma(\mathsf{U}) \cap \gamma(\mathsf{U}') \neq \emptyset$.

4.3 Functions and Abstraction

When typechecking a function application, we must handle the case where the function has type ?. Since ? is not an arrow type, the static version of the rule would fail in all such cases. Instead, we extract the domain and codomain from the type using partial functions. Statically, $\mathbf{dom}\ (x : \mathsf{U}) \to \mathsf{U}' = \mathsf{U}$, and is undefined otherwise. But what should the domain of ? be?

AGT gives a recipe for lifting such gradual functions. To do so, we need the counterpart to concretization: abstraction. The abstraction function α is defined in Figure 6. It takes a set of static terms, and finds the most precise gradual term that is consistent with the entire set. Now, we are able to take gradual terms to sets of static terms, then back to gradual terms. It is easy to see that $\alpha(\gamma(\mathsf{U})) = \mathsf{U}$. This lets us define our partial functions in terms of their static counterparts: we concretize the inputs, apply the static function element-wise on all values of the concretization for which the function is defined, then abstract the output to obtain a gradual term as a result.

For example, the domain of a gradual term U is $\alpha(\{\, \mathbf{dom}\ \mathsf{U}' \mid \mathsf{U}' \in \gamma(\mathsf{U}) \,\})$, which can be expressed syntactically using the rules in Figure 6. We define function-type codomains and lambda-term bodies similarly, though we pair these operations with substitution to avoid creating a "dummy" bound variable name for ?.

Taken together, α and γ form a *Galois-connection*, which ensures that our derived type system is a conservative extension of the static system.

4.4 Typing Rules

Given concretization and abstraction, AGT gives a recipe for converting a static type system into a gradual one, and we follow it closely. Figure 7 gives the rules for typing. Equalities implied by repeated metavariables have been replaced by consistency, such as in GCheckSynth. Similarly, in GCheckPi we use the judgment $\mathsf{U} \cong \mathbf{Set}$ to ensure that the given type is consistent to, rather than equal to, some $\mathbf{Set}_i$. Rules that matched on the form of a synthesized type instead use partial

$\boxed{\Gamma \vdash t \Rightarrow U \quad | \quad \Gamma \vdash t \Leftarrow U}$ *(Well Typed Gradual Terms)*

$$\textsc{GSynthApp}\ \dfrac{\Gamma \vdash t_1 \Rightarrow U \qquad \Gamma \vdash u \rightsquigarrow t_2 \Leftarrow \mathbf{dom}\ U \qquad [u/\_]\mathbf{cod}\ U = U_2}{\Gamma \vdash t_1\ t_2 \Rightarrow U_2}$$

$$\textsc{GCheckPi}\ \dfrac{\Gamma \vdash U' \rightsquigarrow T_1 \Leftarrow U \qquad U \cong \mathbf{Set} \qquad \vdash (x : U')\Gamma \qquad (x : U')\Gamma \vdash T_2 \Leftarrow U}{\Gamma \vdash (x : T_1) \to T_2 \Leftarrow U}$$

$$\textsc{GSynthDyn}\ \dfrac{}{\Gamma \vdash \mathbf{?} \Rightarrow \mathbf{?}}$$

$$\textsc{GCheckLamPi}\ \dfrac{\vdash (x : U_1)\Gamma \qquad (x : U_1)\Gamma \vdash t \Leftarrow U_2}{\Gamma \vdash (\lambda x.\, t) \Leftarrow (x : U_1) \to U_2}$$

$$\textsc{GCheckLamDyn}\ \dfrac{\vdash (x : \mathbf{?})\Gamma \qquad (x : \mathbf{?})\Gamma \vdash t \Leftarrow \mathbf{?}}{\Gamma \vdash (\lambda x.\, t) \Leftarrow \mathbf{?}}$$

$$\textsc{GCheckSynth}\ \dfrac{\Gamma \vdash t \Rightarrow U' \qquad U' \cong U}{\Gamma \vdash t \Leftarrow U}$$

Fig. 7. GDTL: Type Checking and Synthesis (select rules)

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash \text{head}\ \mathbf{Nat}\ 0 \Rightarrow \mathbf{Vec}\ \mathbf{Nat}\ 1 \to \mathbf{Nat}} \qquad \dfrac{\textsc{GSynthAnn}\ \dfrac{\textsc{GCheckSynth}\ \dfrac{\dfrac{\vdots}{\Gamma \vdash \mathbf{Nil}\ \mathbf{Nat} \Rightarrow \mathbf{Vec}\ \mathbf{Nat}\ 0} \qquad \dfrac{\vdots}{\mathbf{Vec}\ \mathbf{Nat}\ 0 \ \cong\ \mathbf{Vec}\ \mathbf{Nat}\ \mathbf{?}}}{\Gamma \vdash \mathbf{Nil}\ \mathbf{Nat} \Leftarrow \mathbf{Vec}\ \mathbf{Nat}\ \mathbf{?}} \qquad \vdots}{\Gamma \vdash (\mathbf{Nil}\ \mathbf{Nat}) :: \mathbf{Vec}\ \mathbf{Nat}\ \mathbf{?} \Rightarrow \mathbf{Vec}\ \mathbf{Nat}\ \mathbf{?}} \qquad \dfrac{\vdots}{\mathbf{Vec}\ \mathbf{Nat}\ \mathbf{?} \ \cong\ \mathbf{Vec}\ \mathbf{Nat}\ 1}}{\Gamma \vdash (\mathbf{Nil}\ \mathbf{Nat}) :: \mathbf{Vec}\ \mathbf{Nat}\ \mathbf{?} \Leftarrow \mathbf{Vec}\ \mathbf{Nat}\ 1}\ \textsc{GCheckSynth}}{\Gamma \vdash \text{head}\ \mathbf{Nat}\ 0\ ((\mathbf{Nil}\ \mathbf{Nat}) :: \mathbf{Vec}\ \mathbf{Nat}\ \mathbf{?}) \Rightarrow \mathbf{Nat}}\ \textsc{GSynthApp}$$

Fig. 8. Type Derivation for head of nil

functions, as we can see in GSynthApp. We split the checking of functions into GCheckLamPi and CCheckLamDyn for clarity, but the rules are equivalent to a single rule using partial functions.

We wish to allow the unknown term **?** to replace any term in a program. But what should its type be? By the AGT philosophy, **?** represents all terms, so it should synthesize the abstraction of all inhabited types, which is **?**. We encode this in the rule GSynthDyn. This means that we can use it in any context.

As with the static system, we represent types in canonical form, which makes consistency checking easy. Well-formedness rules (omitted) are derived from the static system in the same way as the gradual type rules. Additionally, the gradual type rules rely on the *gradual normalization* judgments, $\Gamma \vdash t \rightsquigarrow u \Rightarrow U$ and $\Gamma \vdash u \rightsquigarrow t \Leftarrow U$, which we explain in Section 5.3.

4.5 Example: Typechecking head of nil

To illustrate how the GDTL type system works, we explain the typechecking of one example from the introduction. Suppose we have Church-encodings for natural numbers and vectors, where $\Gamma \vdash \text{head} : (\text{A} : \mathbf{Set}_1) \to (\text{n} : \mathbf{Nat}) \to \mathbf{Vec}\ \text{A}\ (\text{n} + 1) \to \text{A}$. In Figure 8, we show the (partial) derivation of $\Gamma \vdash \text{head}\ \mathbf{Nat}\ 0\ ((\mathbf{Nil}\ \mathbf{Nat}) :: \mathbf{Vec}\ \mathbf{Nat}\ \mathbf{?}) \Rightarrow \mathbf{Nat}$.

The key detail here is that the compile-time consistency check lets us compare 0 to **?**, and then **?** to 1, which allows the example to typecheck. Since $0 \cong \mathbf{?}$, the check succeeds. Notice how we only check consistency when we switch from checking to synthesis. While this code type-checks, it will fail at runtime. We step through its execution in Section 6.4.

5 GRADUAL HEREDITARY SUBSTITUTION

In the previous example, normalization was used to compute the type of head **Nat** 0, replacing n with 0 in the type of head, normalizing 0 + 1 to 1. This computation is trivial, but not all are.

As we saw in Section 2.3, the type-term overlap in GDTL means that we code that is run during type-checking may fail or diverge.

A potential solution would be to disallow imprecisely typed code in type indices. However, this approach breaks the criteria for a gradually-typed language. In particular, it would result in a language that violates the static gradual guarantee (Section 2.4). The static guarantee means that if a program does not typecheck, the programmer knows that the problem is not the absence of type precision, but that the types present are fundamentally wrong. Increasing precision in multiple places will never cause a program to typecheck where doing so in one place fails.

In this section, we present two versions of gradual substitution. First, we provide *ideal substitution*, which is well defined on all terms, but for which equality is undecidable. Second, we describe *approximate hereditary substitution*, which regains decidability while preserving the gradual guarantee, by producing compile-time canonical forms that are potentially less precise than their runtime counterparts. Thus, we trade precision for a termination guarantee. From this, we build *approximate normalization*, which uses hereditary substitution to avoid non-termination, and avoids dynamic failures by normalizing certain imprecise terms to **?**.

A key insight of this work is that we need separate notions of *compile-time normalization* and *run-time execution*. That is, we use approximate hereditary substitution *only* in our types. Executing our programs at run-time will not lose information, but it also may diverge.

For type-checking, the effect of this substitution is that non-equal terms of the unknown type may be indistinguishable at compile-time. Returning to the example from Section 2.3, the user's faulty factorial-length vector will typecheck, but at type **Vec Nat ?**. Using it will never raise a static error due to its length, but it may raise a runtime error.

5.1 Ideal Substitution

Here, we present a definition of gradual substitution for $\alpha\beta\eta$-equivalence classes of terms. While comparing equivalence classes is undecidable, we will use ideal substitution as the theoretical foundation, showing that our approximate substitution produces the same results as ideal substitution, but for some loss of precision.

The main difficulty with lifting the definition of hereditary substitution is that the set of terms with a canonical form is only closed under hereditary substitution when we assume a static type discipline. The terms $y\ y$ and $\lambda x.\ x\ x$ are both syntactically canonical, but if we substitute the second in for y, there is no normal form. However, both of these terms can be typed in our gradual system. How can $[(\lambda x.\ x\ x)/y]^{?}y\ y$ be defined?

If we apply the AGT lifting recipe to hereditary substitution, we get a function that may not have a defined output for all gradually well-typed canonical inputs. Even worse is that determining whether substitution is defined for an input is undecidable. By AGT's formulation, $[\mathrm{u}/x]^{?}\mathrm{u}'$ would be $\alpha(\{\, [\mathrm{u}_1/x]^{\mathrm{U}}\mathrm{u}_2 \mid \mathrm{u}_1 \in \gamma(\mathrm{u}), \mathrm{u}_2 \in \gamma(\mathrm{u}'), \mathrm{U} \in \textsc{SCanonical}\,\})$. To compute α, we must know which of the concretized results are defined, i.e. finding all pairs in $\gamma(\mathrm{u}) \times \gamma(\mathrm{u}')$ for which there exists some U on which static hereditary substitution is defined. This means determining if there is any finite number of substitutions under which the substitution on a (possibly dynamically-typed) term is defined, which requires solving the Halting Problem.

Recall that we introduced canonical forms in Section 3.2 to uniquely represent $\alpha\beta\eta$-equivalence classes. While canonical forms are not closed under substitutions, equivalence classes are. Going back to our initial example, what we really want is for $[(\lambda x.\ x\ x)/y]^{?}y\ y$ to be $((\lambda x.\ x\ x)(\lambda x.\ x\ x))^{\alpha\beta\eta}$, i.e. the set of all terms $\alpha\beta\eta$-equivalent to Ω.

Instead we define ideal substitution on $\alpha\beta\eta$-equivalence classes themselves. For this, we do not need hereditary substitution: if $s \in s^{\alpha\beta\eta}$ and $t \in t^{\alpha\beta\eta}$ are terms with their respective equivalence

$\boxed{[\mathrm{u}/x]^{\mathrm{U}}\, x\overline{\mathrm{s}} = \mathrm{u}_2 : \mathrm{U}_2}$ ***(Approximate Atomic Hereditary Substitution)***

$$\textsc{GHsubRHead}\quad \frac{}{[\mathrm{u}/x]^{\mathrm{U}}\, x = \mathrm{u} : \mathrm{U}}$$

$$\textsc{GHsubRDynSpine}\quad \frac{[\mathrm{u}/x]^{\mathrm{U}}\, x\overline{\mathrm{s}} = ? : (y : \mathrm{U}_1) \to \mathrm{U}_2 \qquad [\mathrm{u}_2/y]^{\mathrm{U}_1}\mathrm{U}_2 = \mathrm{U}_3}{[\mathrm{u}/x]^{\mathrm{U}}\, x\overline{\mathrm{s}}\ \mathrm{u}_2 = ? : \mathrm{U}_3}$$

$$\textsc{GHsubRLamSpine}\quad \frac{\begin{array}{cc} [\mathrm{u}/x]^{\mathrm{U}}\, x\overline{\mathrm{s}} = (\lambda y.\, \mathrm{u}_2) : (y : \mathrm{U}_1) \to \mathrm{U}_2 & [\mathrm{u}/x]^{\mathrm{U}}\, \mathrm{u}_1 = \mathrm{u}_3 \\ [\mathrm{u}_3/y]^{\mathrm{U}_1}\mathrm{u}_2 = \mathrm{u}_4 \qquad [\mathrm{u}_3/y]^{\mathrm{U}_1}\mathrm{U}_2 = \mathrm{U}_3 & \mathrm{u}_4 \rightsquigarrow_\eta \mathrm{u}_5 : \mathrm{U}_3 \end{array}}{[\mathrm{u}/x]^{\mathrm{U}}\, x\overline{\mathrm{s}}\ \mathrm{u}_1 = \mathrm{u}_5 : \mathrm{U}_3}$$

$$\textsc{GHsubRDynType}\quad \frac{[\mathrm{u}_1/x]^{\mathrm{U}}\, x\overline{\mathrm{s}} = \mathrm{u}_2 : ?}{[\mathrm{u}_1/x]^{\mathrm{U}}\, x\overline{\mathrm{s}}\ \mathrm{u}_2 = ? : ?}$$

Fig. 9. GDTL: Approximate Substitution (select rules)

classes, the substitution $[x \Mapsto s^{\alpha\beta\eta}]t^{\alpha\beta\eta}$ is simply the equivalence class of $[x \Mapsto s]t$. We now have a total operation from equivalence classes to equivalence classes. These classes may have no canonical representative, but the function is defined regardless. If we extend concretization and abstraction to be defined on equivalence classes, this gives us the definition of ideal substitution:

$$[x \Mapsto \mathrm{t}_1{}^{\alpha\beta\eta}]\mathrm{t}_2{}^{\alpha\beta\eta} = \alpha(\{[x \Mapsto \mathrm{t}_1']\mathrm{t}_2' \mid \mathrm{t}_1' \in \gamma(\mathrm{t}_1), \mathrm{t}_2' \in \gamma(\mathrm{t}_2)\})$$

That is, we first find the concretization of the gradual equivalence classes, which are sets of static equivalence classes. We then compute the substitution of each combination of these by taking the substitution of a representative element. We finally abstract over this set to obtain a single gradual equivalence class.

5.2 Approximate Substitution

With a well-defined but undecidable substitution, we now turn to the problem of how to recover decidable equality for equivalence classes, without losing the gradual guarantees. We again turn to (gradual) canonical forms as representatives of $\alpha\beta\eta$-equivalence classes. What happens when we try to construct a hereditary substitution function syntactically, as in SDTL?

The problem is in adapting SHsubRSpine. Suppose we are substituting u for x in $x\overline{\mathrm{s}}\ \mathrm{u}_2$, and the result of substituting in $x\overline{\mathrm{s}}$ is $(\lambda y.\, \mathrm{u}') : ?$. Following the AGT approach, we can use the **dom** function to calculate the domain of $?$, which is the type at which we will substitute y. But this violates the well-foundedness condition we imposed in the static case! Since the domain of $?$ is $?$, eliminating redexes may infinitely apply substitutions without decreasing the size of the type.

In all other cases, we have no problem, since the term we are substituting into is structurally decreasing. So, while equivalence classes give us our ideal, theoretical definition, hereditary substitution provides us with the exact cases we must approximate in order to preserve decidability. To guarantee termination, we must not perform recursive substitutions in spines with $?$-typed heads.

There are two obvious options for how to proceed without making recursive calls: we either fail when we try to apply a $?$-typed function, or we return $?$. The former will preserve termination, but it will not preserve the static gradual guarantee. Reducing the precision of a well-typed program's ascriptions should never yield ill-typed code. If applying a dynamically-typed function caused failure, then changing an ascription to $?$ could cause a previously successful program to crash, violating the guarantee.

$\boxed{\Gamma \vdash t \rightsquigarrow u \Rightarrow U \quad | \quad \Gamma \vdash u \leftsquigarrow t \Leftarrow U}$ *(**Approximate Normalization**)*

$$\textsc{GNSynthApp}\quad \frac{\begin{array}{cc}\Gamma \vdash t_1 \rightsquigarrow u_1 \Rightarrow U & u_1 \rightsquigarrow_\eta u_1' : ? \to ? \\ \mathbf{dom}\ U = U_1 & \Gamma \vdash u_2 \leftsquigarrow t_2 \Leftarrow U_1 \\ \multicolumn{2}{c}{[u_2/\_]^{U_1}\mathbf{body}\ u' = u_3} \\ [u_2/\_]\mathbf{cod}\ U = U_2 & u_3 \rightsquigarrow_\eta u_3' : U_2\end{array}}{\Gamma \vdash t_1\ t_2 \rightsquigarrow u_3' \Rightarrow U_2}$$

$$\textsc{GNCheckSynth}\quad \frac{\begin{array}{c}\Gamma \vdash t \rightsquigarrow u \Rightarrow U' \\ U \cong U' \quad U' \sqsubseteq U\end{array}}{\Gamma \vdash u \leftsquigarrow t \Leftarrow U} \qquad \textsc{GNCheckApprox}\quad \frac{\begin{array}{c}\Gamma \vdash t \rightsquigarrow u \Rightarrow U' \\ U \cong U' \quad U' \not\sqsubseteq U\end{array}}{\Gamma \vdash ? \leftsquigarrow t \Leftarrow U}$$

Fig. 10. GDTL: Approximate Normalization (select rules)

Our solution is to produce ? when applying a function of type ?. We highlight the changes to hereditary substitution in Figure 9. GHsubRDynType accounts for ?-typed functions, and GHsubRDynSpine accounts for ? applied as a function.

5.3 Approximate Normalization

While approximate hereditary substitution eliminates non-termination, we must still account for dynamic failures. We do so with *approximate normalization* (Figure 10).

To see the issue, consider that we can type the term $(\lambda A.\,(0 :: ?) :: A)$ against $(A : \mathbf{Set}_1) \to A$. However, there are no ascriptions in canonical form, since ascriptions can induce casts, which are a form of computation. The term $(\lambda A.\,0)$ certainly does not type against $(A : \mathbf{Set}_1) \to A$, since 0 will not check against the type variable A. However, if we were to raise a type error, we would never be able to apply a function to the above term. In the context $(A : \mathbf{Set}_1)\cdot$, the only canonical term with type A is ?. So if the term is to have a well-typed normal form, its body must be ?.

More broadly, normalization does not preserve *synthesis* of typing, only checking. In GNCheckSynth, if $\Gamma \vdash t \Rightarrow U'$, then the normal form of t might check against U, but it won't necessarily synthesize U (or any type). We need to construct a canonical form u for t at type U, assuming we have some normal form u' for t at type U'. If $U \sqsubseteq U'$, u' will check against U. But otherwise, the only value we can be sure will check against U is ?, which checks against any type. We formalize this using a pair of normalization rules: GNCheckSynth normalizes fully when we can do so in a type-safe way, and GNCheckApprox produces ? as an approximate result in all other cases.

Gradual types must treat η-expansion carefully. We η-expand all variables in GNSynthVar, but in GNSynthVar, we may be applying a function of type ?. However, a variable x of type ? is η-long. Since we are essentially treating a value of type ? as type $? \to ?$, we must expand x to $(\lambda y.\, x\ y)$. We do this in GNSynthVar with an extra η-expansion at type $? \to ?$, which expands a ?-typed term one level, and has no effect on a canonical form with a function type.

The remaining rules for normalization (omitted) directly mirror the rules from Figure 7. $\mathbf{Set}_i$, ? and variables all normalize to themselves, and all other rules simply construct normal forms from the normal forms of their subterms.

Some of the difficulty with normalization comes from the fact that function arguments are normalized before being substituted. One could imagine a language that normalizes after substituting function arguments, and typechecking fails if a dynamic error is encountered during normalization. Here, normalization could fail, but only on terms that had truly ill-formed types, since unused failing values would be discarded. We leave the development of such a language to future work.

5.4 Properties of Approximate Normalization

Relationship to the Ideal. If we expand our definition of concretization to apply to equivalence classes of terms, gradual precision gives us a formal relationship between ideal and approximate normalization:

Theorem 5.1 (Normalization Approximates the Ideal). *For any* Γ, t, U, *if* $\Gamma \vdash t \Leftarrow U$, *then* $\Gamma \vdash u \leftrightsquigarrow t \Leftarrow U$ *for some* u, *and* $t^{\alpha\beta\eta} \sqsubseteq u$.

Intuitively, this holds because approximate normalization for a term either matches the ideal, or produces ?, which is less precise than every other term.

Preservation of Typing. To prove type soundness for GDTL, a key property of normalization is that it preserves typing. This property relies on the fact that hereditary substitution preserves typing, which can be shown using a technique similar to that of Pfenning [2008].

Theorem 5.2 (Normalization preserves typing). *If* $\Gamma \vdash u \leftrightsquigarrow t \Leftarrow U$, *then* $\Gamma \vdash u \Leftarrow U$.

Normalization as a Total Function. Since we have defined our substitution and normalization using inference rules, they are technically relations rather than functions. Since the rules are syntax directed in terms of their inputs, it is fairly easy to show that there is at most one result for every set of inputs. As we discussed above, the approximation in GNCheckApprox allows normalization to be total.

Theorem 5.3 (Normalization is Total). *If* $\Gamma \vdash t \Leftarrow U$, *then* $\Gamma \vdash u \leftrightsquigarrow t \Leftarrow U$ *for exactly one* u.

6 GDTL: RUNTIME SEMANTICS

With the type system for GDTL realized, we turn to its dynamic semantics. Following the approaches of Garcia et al. [2016] and Toro et al. [2018b], we let the syntactic type-safety proof for the static SDTL drive its design. In place of a cast calculus, gradual terms carry *evidence* that they match their type, and computation steps evolve that evidence incrementally. When evidence no longer supports the well-typedness of a term, execution fails with a runtime type error.

6.1 The Runtime Language

Figure 11 gives the syntax for our runtime language. It largely mirrors the syntax for gradual terms, with three main changes. In place of type ascriptions, we have a special form for terms augmented with evidence, following Toro et al. [2018b]. We also have err, an explicit term for runtime type errors.

Translation proceeds by augmenting our bidirectional typing rules to output the translated term. Type ascriptions are dropped in the GSynthAnn rule, and initial evidence of consistency is added in GCheckSynth. Section 6.2 describes how to derive this initial evidence. In the GSynthDyn rule, we annotate ? with evidence ?, so ? is always accompanied by some evidence of its type.

In Figure 11 we also define the class of syntactic values, which determines those terms that are done evaluating. We wish to allow values to be augmented with evidence, but not to have multiple evidence objects stacked on a value. To express this, we separate the class of values from the class of *raw values*, which are never ascribed with evidence at the top level.

Values are similar to, but not the same, as canonical forms. In particular, there are no redexes in canonical terms, even beneath a λ, whereas values may contain redexes within abstractions.

6.2 Typing and Evidence

To establish progress and preservation, we need typing rules for evidence terms, whose key rules we highlight in Figure 11. These are essentially same as for gradual terms, with two major changes.

Evidence Terms

$$\begin{array}{rcl} e, E & ::= & \\ & | & \lambda x.\, e \\ & | & e_1\ e_2 \\ & | & x \\ & | & (x : E_1) \rightarrow E_2 \\ & | & \mathbf{Set}_i \\ & | & \mathbf{?} \\ & | & \varepsilon\, e \\ & | & \mathsf{err} \end{array}$$

Evidence Values

$$\begin{array}{rcl} v, V & ::= & \\ & | & \varepsilon\, w \\ & | & w \end{array}$$

Raw Values

$$\begin{array}{rcl} w, W & ::= & \\ & | & \lambda x.\, e \\ & | & (x : V) \rightarrow E \\ & | & \mathbf{Set}_i \\ & | & \mathbf{?} \end{array}$$

Evidence Contexts

$$\begin{array}{rcl} C & ::= & \\ & | & \square\, e \\ & | & v\, \square \\ & | & (x : \square) \rightarrow E \\ & | & \varepsilon\, \square \end{array}$$

$\boxed{\Gamma \vdash e : U}$ ***(Evidence Term Typing)***

$$\textsc{EvTypeApp}\ \frac{\Gamma \vdash e_1 : U \quad \Gamma \vdash e_2 : \mathbf{dom}\ U \quad [e_2/\_]\mathbf{cod}\ U = U_2}{\Gamma \vdash e_1\ e_2 : U_2}$$

$$\textsc{EvTypeEv}\ \frac{\Gamma \vdash e : U' \quad \varepsilon \vdash U' \cong U}{\Gamma \vdash \varepsilon\, e : U}$$

$$\textsc{EvTypeDyn}\ \frac{\Gamma \vdash U : \mathbf{Set} \quad \varepsilon \vdash U \cong U}{\Gamma \vdash \varepsilon\, \mathbf{?} : U}$$

Fig. 11. Evidence Term Syntax and Typing (select rules)

First, we no longer use bidirectional typing, since our type system need not be syntax directed to prove soundness. Second, we restrict consistency to terms ascribed with evidence, and we require that the evidence supports the consistency of the two types (given by the relation $\varepsilon \vdash U \cong U$). This means all applications of consistency are made explicit in the syntax of evidence terms. We also have a notion of normalization for evidence terms, which erases evidence ascriptions and otherwise behaves like the normalization for gradual terms. This normalization allows us to define hereditary substitutions of evidence terms into types.

This raises the question: what is evidence? AGT provides a highly general approach, applicable to multi-argument, asymmetric predicates. However, since equality is the only predicate we use, the meet of two terms is sufficient to serve as evidence of their consistency. We say that $\langle U' \rangle \vdash U_1 \cong U_2$ whenever $U' \sqsubseteq U_1 \sqcap U_2$. If a term synthesizes some U and is checked against U', then during elaboration we can ascribe to it the evidence $U \sqcap U'$. Similarly, if $\langle U \rangle \vdash U_1 \cong U_2$, and $\langle U' \rangle \vdash U_2 \cong U_3$, then $\langle U \sqcap U' \rangle \vdash U_1 \cong U_3$. The meet operator fills the roles that Garcia et al. [2016] refer to as *initial evidence* and *consistent transitivity*, respectively.

Initial evidence for $U \cong U'$must take the form $\langle \Gamma, U'' \rangle$. This environment stores the types of variables that were free in the derivation of $U \cong U'$. Since types and terms overlap, evidence in terms under binders may contain free variables. When we substitute their values using hereditary substitution, we require the types of the variables. We write $\langle U \rangle$ as a shorthand when the environment is empty. When the environment for evidence objects are empty, we freely use them in type operations (e.g. $\varepsilon_1 \sqcap \varepsilon_2$).

6.3 Developing a Safe Semantics

To devise our semantics, we imagine a hypothetical proof of progress and preservation. Progress tells us which expressions we need reduction rules for, and preservation tells us how to step in order to remain well typed.

$\boxed{e_1 \longrightarrow e_2}$ ***(Evidence-based Small-Step Semantics)***

$$\textsc{StepAscr}\ \frac{\varepsilon_1 \sqcap \varepsilon_2 = \varepsilon_3}{\varepsilon_1\,(\varepsilon_2\,w) \longrightarrow \varepsilon_3\,w} \qquad \textsc{StepAscrFail}\ \frac{\varepsilon_1 \sqcap \varepsilon_2\ \textbf{undefined}}{\varepsilon_1\,(\varepsilon_2\,w) \longrightarrow \mathsf{err}} \qquad \textsc{StepAppEv}\ \frac{\varepsilon_2 \sqcap \textbf{dom}\ \varepsilon_1 = \varepsilon_3 \quad [w/\_]\textbf{cod}\ \varepsilon_1 = \varepsilon_4}{(\varepsilon_1\,(\lambda x.\,e))\,(\varepsilon_2\,w) \longrightarrow \varepsilon_4\,([x \Mapsto \varepsilon_3\,w]e)}$$

$$\textsc{StepAppEvRaw}\ \frac{\textbf{dom}\ \varepsilon_1 = \varepsilon_2 \quad [w/\_]\textbf{cod}\ \varepsilon_1 = \varepsilon_3}{(\varepsilon_1\,(\lambda x.\,e))\,w \longrightarrow \varepsilon_3\,([x \Mapsto \varepsilon_2\,w]e)} \qquad \textsc{StepApp}\ \frac{}{(\lambda x.\,e)\,v \longrightarrow [x \Mapsto v]e} \qquad \textsc{StepAppDyn}\ \frac{[v/\_]\textbf{cod}\ \varepsilon_1 = \varepsilon_2}{(\varepsilon_1\,?)\,v \longrightarrow \varepsilon_2\,?}$$

$$\textsc{StepAppFailTrans}\ \frac{\textbf{dom}\ \varepsilon_1 = \varepsilon_3 \quad \varepsilon_3 \sqcap \varepsilon_2\ \textbf{undefined}}{(\varepsilon_1\,w_1)\,(\varepsilon_2\,w_2) \longrightarrow \mathsf{err}} \qquad \textsc{StepAppFailDom}\ \frac{\textbf{dom}\ \varepsilon_1\ \textbf{undefined}}{(\varepsilon_1\,w)\,v \longrightarrow \mathsf{err}} \qquad \textsc{StepContext}\ \frac{e_1 \longrightarrow e_2 \quad e_1, e_2 \neq \mathsf{err}}{C[e_1] \longrightarrow C[e_2]} \qquad \textsc{StepContextErr}\ \frac{e \longrightarrow \mathsf{err}}{C[e] \longrightarrow \mathsf{err}}$$

Fig. 12. GDTL: Dynamic Semantics

Double Evidence. Since our values do not contain terms of the form $\varepsilon_2\,(\varepsilon_1\,w)$, progress dictates that we need a reduction rule for such a case. If $\cdot \vdash w : U$, $\varepsilon_1 \vdash U \cong U'$ and $\varepsilon_2 \vdash U' \cong U''$, then $\varepsilon_1 \sqcap \varepsilon_2 \vdash U \cong U''$, so we step to $(\varepsilon_1 \sqcap \varepsilon_2)w$. If the meet is not defined, then a runtime error occurs.

Functions with Evidence. While our simple rules dictate how to evaluate a λ-term applied to a value, they do not determine how to proceed for applications of the form $(\varepsilon_1\,\lambda x.\,\mathsf{es})\,(\varepsilon_2\,w)$.

In such a case, we know that $\cdot \vdash (\varepsilon_1\,\lambda x.\,\mathsf{es}) : U$ and that $\varepsilon_1 \vdash U \cong U'$ for some U'. Computing $\textbf{dom}\ \varepsilon_1 \sqcap \varepsilon_2$ yields evidence that w is consistent with the domain of U, so we ascribe this evidence during substitution to preserve well-typedness. Our evidence-typing rules say that the type of an application is found by normalizing the argument value and substituting into the codomain of the function type. To produce a result at this type, we can normalize w and substitute it into the codomain of ε_1, thereby producing evidence that the actual result is consistent with the return type.

In the case where w is not ascribed with evidence, we can simply ascribe it $\langle ? \rangle$ and proceed using the above process.

Applying The Unknown Term. Our syntax for values only admits application under binders, so we must somehow reduce terms of the form $(\varepsilon\,?)\,v$. The solution is simple: if the function is unknown, so is its output.

Since the unknown term is always accompanied by type evidence at runtime, we calculate the result type by substituting the argument into the codomain of the evidence associated with $?$.

The Full Semantics. All other well-typed terms are either values, or contain a redex as a subterm, either of the simple variety or of the varieties described above. Using contextual-rules to account for these remaining cases, we have a semantics that satisfies progress and preservation *by construction.* Figure 12 gives the full set of rules.

6.4 Example: Running head of nil

We return to the example from Section 4.5, this time explaining its runtime behaviour. Because of consistency, the term (**Nil Nat** :: (**Vec Nat** ?)) is given the evidence $\langle$**Vec Nat** ?$\rangle$ (i.e. $\langle$**Vec Nat** $0 \sqcap$ **Vec Nat** ?$\rangle$). Applying consistency to use this as an argument adds the evidence $\langle$**Vec Nat** $1\rangle$, since we check (**Nil Nat** :: (**Vec Nat ?**)) against **dom** (**Vec Nat** 1 ⟶ **Nat**). The rule StepContext dictates that we must evaluate the argument to a function before evaluation the application itself. Our argument is ⟨**Vec Nat** 1⟩⟨**Vec Nat** 0⟩(**Nil Nat**), and since the meet of the evidence types is undefined, we step to err with StepAscrFail.

7 PROPERTIES OF GDTL

GDTL satisfies all the criteria for gradual languages set forth by Siek et al. [2015].

Safety. First, GDTL is type sound *by construction*; the runtime semantics are specifically crafted to maintain progress and preservation. We can then obtain the standard safety result for gradual languages, namely that well-typed terms do not get stuck.

Theorem 7.1 (Type safety). *If* $\cdot \vdash e : U$*, then either* $e \longrightarrow^* v$ *for some* v*,* $e \longrightarrow^*$ err*, or* e *diverges.*

Conservative Extension of SDTL. It is easy to show that GDTL is a conservative extension of SDTL. This means that any fully-precise GDTL programs enjoy the soundness and consistency properties that SDTL guarantees. Any statically-typed term is well-typed in GDTL by construction, thanks to AGT: on fully precise gradual types, $\alpha \circ \gamma$ is the identity. Moreover, the *only* additions are those pertaining to **?**, meaning that if we restrict ourself to the static subset of terms (and types) without **?**, then we have all the properties of the static system. We formalize this as follows:

Theorem 7.2. *If* Γ, t, U *are the embeddings of some* Γ, t, U *into GDTL, and* $\Gamma \vdash t \Leftarrow U$*, then* $\Gamma \vdash t \Leftarrow U$*. Moreover, there exists some* e*,* v *and* v *where* $e \longrightarrow^* v$*,* e *is the embedding of* t *into evidence terms, and* v *is* v *embedded into evidence terms.*

Embedding of Untyped Lambda Calculus. A significant property of GDTL is that it can fully embed the untyped lambda calculus, including non-terminating terms. Given an untyped embedding function $\lceil t \rceil$ that (in essence) annotates all terms with **?** we can show that any untyped term can be embedded in our system. Since no type information is present, all evidence objects are **?**, meaning that the meet operator never fails and untyped programs behave normally in GDTL.

Theorem 7.3. *For any term* t*, if* Γ *maps all variables to type ?, then* $\Gamma \vdash \lceil t \rceil \Rightarrow ?$*.*

Gradual Guarantees. GDTL smoothly supports the full spectrum in between dependent and untyped programming—a property known as the gradual guarantee [Siek et al. 2015], which comes in two parts:

Theorem 7.4 (Gradual Guarantee).
(Static Guarantee) Suppose $\Gamma \vdash t \Leftarrow U$ *and* $U \sqsubseteq U'$*. If* $\Gamma \sqsubseteq \Gamma'$ *and* $t \sqsubseteq t'$*, then* $\Gamma_2 \vdash t' \Leftarrow U'$*.*
(Dynamic Guarantee) Suppose that $\cdot \vdash e_1 : U$*,* $\cdot \vdash e_1' : U'$*,* $e_1 \sqsubseteq e_1'$*, and* $U \sqsubseteq U'$*. Then if* $e_1 \longrightarrow^* e_2$*, then* $e_1' \longrightarrow^* e_2'$ *where* $e_2 \sqsubseteq e_2'$*.*

AGT ensures that the gradual guarantee holds by construction. Specifically, because approximate normalization and consistent transitivity are both monotone with respect to the precision relation, we can establish a weak bisimulation between the steps of the more and less precise versions [Garcia et al. 2016].

A novel insight that arises from our work is that we need a restricted form of the dynamic gradual guarantee *for normalization* in order to prove the static gradual guarantee. To differentiate it from the standard one, we call it the *normalization gradual guarantee*. Because an η-long term might be longer at a more precise type, we phrase the guarantee modulo η-equivalence.

Lemma 7.5 (Normalization Gradual Guarantee). *Suppose* $\Gamma_1 \vdash u_1 \leftsquigarrow t_1 \Leftarrow U_1$*. If* $\Gamma_1 \sqsubseteq^\eta \Gamma_2$*,* $t_1 \sqsubseteq t_2$*, and* $U_1 \sqsubseteq^\eta U_2$*, then* $\Gamma_2 \vdash u_2 \leftsquigarrow t_2 \Leftarrow U_2$ *where* $u_1 \sqsubseteq^\eta u_2$*.*

$$
\begin{array}{rl}
\mathbf{Nat}: & \mathbf{Set}_1 \\
\mathbf{Vec}: & \mathbf{Set}_i \to \mathbf{Nat} \to \mathbf{Set}_i \\
\mathbf{Eq}: & (\mathsf{A} : \mathbf{Set}_i) \to \mathsf{A} \to \mathsf{A} \to \mathbf{Set}_i
\end{array}
\qquad
\begin{array}{rl}
0: & \mathbf{Nat} \\
\mathbf{Succ}: & \mathbf{Nat} \to \mathbf{Nat} \\
\mathbf{Nil}: & (\mathsf{A} : \mathbf{Set}_i) \to \mathbf{Vec}\ \mathsf{A}\ 0 \\
\mathbf{Refl}: & (\mathsf{A} : \mathbf{Set}_i) \to (x : \mathsf{A}) \to \mathbf{Eq}\ \mathsf{A}\ x\ x
\end{array}
$$

$$
\begin{array}{rl}
\mathbf{Cons}: & (\mathsf{A} : \mathbf{Set}_i) \to (\mathsf{n} : \mathbf{Nat}) \to (\mathsf{hd} : \mathsf{A}) \to (\mathtt{tl} : \mathbf{Vec}\ \mathsf{A}\ \mathsf{n}) \to \mathbf{Vec}\ \mathsf{A}\ (\mathsf{n}+1) \\
\mathbf{natElim}: & (\mathsf{m} : (\mathbf{Nat} \to \mathbf{Set}_i)) \to (\mathsf{m}\ 0) \to ((\mathsf{n} : \mathbf{Nat}) \to \mathsf{m}\ \mathsf{n} \to \mathsf{m}\ (\mathsf{n}+1)) \\
& \to (\mathsf{n} : \mathbf{Nat}) \to \mathsf{m}\ \mathsf{n} \\
\mathbf{vecElim}: & (\mathsf{A} : \mathbf{Set}_i) \to (\mathsf{n} : \mathbf{Nat}) \to (\mathsf{m} : (\mathbf{Vec}\ \mathsf{A}\ \mathsf{N} \to \mathbf{Set}_i)) \to \mathsf{m}\ (\mathbf{Vec}\ \mathsf{A}\ 0) \\
& \to ((\mathsf{n}_2 : \mathbf{Nat}) \to (\mathsf{h} : \mathsf{A}) \to (\mathtt{tl} : \mathbf{Vec}\ \mathsf{A}\ \mathsf{n}_2) \to \mathsf{m}\ \mathsf{vec} \to \mathsf{m}\ (\mathbf{Cons}\ \mathsf{A}\ (\mathsf{n}_2+1)\ \mathsf{hd}\ \mathtt{tl})) \\
& \to (\mathsf{vec} : \mathbf{Vec}\ \mathsf{A}\ \mathsf{n}) \to \mathsf{m}\ \mathsf{vec} \\
\mathbf{eqElim}: & (\mathsf{A} : \mathbf{Set}_i) \to (\mathsf{m} : (x : \mathsf{A}) \to (y : \mathsf{A}) \to \mathbf{Eq}\ \mathsf{A}\ x\ y \to \mathbf{Set}_i) \to ((z : \mathsf{A}) \to \mathsf{m}\ z\ z\ (\mathbf{Refl}\ \mathsf{A}\ z)) \\
& \to (x : \mathsf{A}) \to (y : \mathsf{A}) \to (\mathsf{p} : \mathbf{Eq}\ \mathsf{A}\ x\ y) \to (\mathsf{m}\ x\ y\ \mathsf{p})
\end{array}
$$

Fig. 13. Constructor and Eliminator Types

8 EXTENSION: INDUCTIVE TYPES

Though GDTL provides type safety and the gradual guarantees, its lack of inductive types means that programming in cumbersome, requiring Church encodings. Additionally, induction principles, along with basic facts like $0 \neq 1$, cannot be proven in the purely negative Calculus of Constructions [Stump 2017]. However, we can type such a term if we introduce inductive types with eliminators, and allow types to be defined in terms of such eliminations.

This section describes how to extend GDTL with a few common inductive types—natural numbers, vectors, and an identity type for equality proofs— along with their eliminators. While not as useful as user-defined types or pattern matching (both of which are important subjects for future work), this specific development illustrates how our approach can be extended to a more full-fledged dependently-typed language. Note that while we show how inductives can be added to the language, extending our metatheory to include inductives is left as future work.

Syntax and Typing. To accommodate natural numbers and vectors, we augment the syntax for terms as follows:

$$
\begin{array}{rl}
t, T ::= & \ldots \mid \mathbf{Nat} \mid 0 \mid t+1 \mid \mathbf{Vec}\ T\ t \mid \mathbf{Eq}\ T\ t_1\ t_2 \mid \mathbf{Refl}\ T\ t \mid \mathbf{Nil}\ \mathsf{gs} \mid \mathbf{Cons}\ T\ t_1\ t_2\ t_3 \\
& \mid \mathbf{natElim}\ T\ t_1\ t_2\ t_3 \mid \mathbf{vecElim}\ T_1\ t_1\ T_2\ t_2\ t_3\ t_4 \mid \mathbf{eqElim}\ T_1\ T_2\ t_1\ t_2\ t_3\ t_4
\end{array}
$$

The typing rules are generally straightforward. We omit the full rules, but we essentially type them as functions that must be fully applied, with the types given in Figure 13.

Each form checks its arguments against the specified types, and the rule GCheckSynth ensures that typechecking succeeds so long as argument types are consistent with the expected types.

Adding these constructs to canonical forms is interesting. Specifically, the introduction forms are added as atomic forms, and the eliminators become new variants of the canonical spines. Since **natElim** applied to a **Nat** is a redex, canonical forms can eliminate variables.

$$\mathrm{r}, \mathrm{R} ::= \quad \ldots \mid \mathbf{Nat} \mid 0 \mid \mathrm{u} + 1 \mid \text{etc.}\ldots$$
$$\bar{\mathrm{s}} ::= \quad \ldots \mid \mathbf{natElim}\ \mathrm{u}_1\ \mathrm{u}_2\ \mathrm{u}_3 \mid \mathbf{vecElim}\ \mathrm{U}_1\ \mathrm{u}_1\ \mathrm{U}_2\ \mathrm{u}_2\ \mathrm{u}_3 \mid \text{etc.}\ldots$$

Notice that the eliminators each take one fewer argument than in the term version, since the last argument is always the rest of the spine in which the eliminator occurs.

Normalization. We extend hereditary substitution for inductive types. For introduction forms, we simply substitute in the subterms. For eliminators, if we are ever replacing x with u' in $x\bar{\mathrm{s}}$ **natElim** $\mathrm{u}_1\ \mathrm{u}_2\ \mathrm{u}_3$, then we substitute in $x\bar{\mathrm{s}}$ and see if we get 0 or **Succ** back. If we get 0, we produce u_2, and if we get $\mathtt{n} + 1$, we compute the recursive elimination for $\mathtt{n}$ as u_2', and substitute $\mathtt{n}$ and u_2' for the arguments of u_3. Vectors and are handled similarly. An application of **eqElim** to **Refl** $\mathtt{A}\ x$ simply returns the given value of type $\mathtt{m}\ x\ x$ (**Refl** $x\ \mathtt{A}$) as a value of type $\mathtt{m}\ x\ y\ \mathtt{p}$.

How should we treat eliminations with **?** as a head? Since **?** represents the set of all static values, the result of eliminating it is the abstraction of the eliminations of all possible values. Since these values may produce conflicting results, the abstraction is simply **?**, which is our result. However, for equality, we have a special case. Each instance of **eqElim** can have only one possible result: the given value, considered as having the output type. Then, we abstract a singleton set. This means we can treat each application of **eqElim** to **?** as an application to **Refl ? ?**. This principle holds for any single-constructor inductive type.

With only functions, we needed to return **?** any time we applied a dynamically-typed function. However, with eliminators, we are always structurally decreasing on the value being eliminated. For **natElim**, we can eliminate a **?**-typed value provided it is 0 or **Succ** u, but otherwise we must produce **?** for substitution to be total and type-preserving. The other inductives are handled in the same way.

Runtime Semantics. The runtime semantics are fairly straightforward. Eliminations are handled as with hereditary substitution: eliminating **?** produces **?**, except with **Eq**, where we treat an elimination on **?** as an elimination on **Refl ? ?**. When applying eliminators or constructor applications, evidence is composed as with functions.

One advantage of GDTL is that the meet operator on evidence allows us to compare values at runtime. Thus, if we write **Refl ? ?**, but use it at type **Eq** $\mathtt{A}\ x\ y$, then it is ascribed with evidence $\langle$**Eq** $\mathtt{A}\ x\ y\rangle$. If we ever use this proof to transform a value of type $\mathtt{P}\ x$ into one of type $\mathtt{P}\ y$, the meet operation on the evidence ensures that the result actually has type $\mathtt{P}\ y$.

Returning to the $\mathtt{head}'$ function from Section 2.3, in $\mathtt{head}'$ **Nat** 0 **? ?** `staticNil`, the second **?** is ascribed with the evidence $\langle$**Eq Nat** $0\ (\mathbf{?} + 1)\rangle$. The call to `rewrite` using this proof tries to convert a **Vec** of length 0 into one of length 1, which adds the evidence $\langle$**Eq Nat** $0\ 1\rangle$ to our proof term. Evaluation tries to compose the layered evidence, but fails with the rule StepAscrFail, since they cannot be composed.

9 RELATED WORK

SDTL. The static dependently-typed language SDTL from which GDTL is derived incorporates many features and techniques from the literature. The core of the language is very similar to that of CC_ω [Coquand and Huet 1988], albeit without an impredicative Prop sort. The core language of Idris [Brady 2013], TT, also features cumulative universes with a single syntactic category for terms and types. Our use of canonical forms draws heavily from work on the Logical Framework

(LF)[Harper et al. 1993; Harper and Licata 2007]. The bidirectional type system we adopt is inspired by Löh et al. [2010]'s tutorial.

Our formulation of hereditary substitution [Pfenning 2008; Watkins et al. 2003] in SDTL is largely drawn from that of Harper and Licata [2007], particularly the type-outputting judgment for substitution on atomic forms, and the treatment of the variable type as an extrinsic argument.

Mixing Dependent Types and Nontermination. Dependently-typed languages that admit nontermination either give up on logical consistency altogether (Ωmega [Sheard and Linger 2007], Haskell), or isolate a sublanguage of terminating expressions. This separation can be either enforced through the type system (Aura [Jia et al. 2008], F★ [Swamy et al. 2016]), a termination checker (Idris [Brady 2013]), or through a strict syntactic separation (Dependent ML [Xi and Pfenning 1999], ATS [Chen and Xi 2005]). The design space is very wide, reflecting a variety of sensible tradeoffs between expressiveness, guarantees, and flexibility.

The Zombie language [Casinghino et al. 2014; Sjöberg et al. 2012] implements a flexible combination of programming and proving. The language is defined in two separate, but interacting, fragments: the programmatic fragment ensures type safety but not termination, and the logical fragment (a syntactic subset of the programmatic one) guarantees logical consistency. Programmers must declare in which fragment a given definition lives, but mobile types and cross-fragment case expressions allow interactions between the fragments. Zombie embodies a very different tradeoff from GDTL: while the logical fragment is consistent as a logic, typechecking may diverge due to normalization of terms from the programmatic fragment. In contrast, GDTL eschews logical consistency as soon as imprecision is introduced (with **?**), but approximate normalization ensures that typechecking terminates.

In general, gradual dependent types as provided in GDTL can be interpreted as a smooth, tight integration of such a two-fragment approach. Indeed, the subset of GDTL that is fully precise corresponds to SDTL, which is consistent as a logic. However, in the gradual setting, the fragment separation is fluid: it is driven by the precision of types and terms, which is free to evolve at an arbitrary fine level of granularity. Also, the mentioned approaches are typically not concerned with accommodating the flexibility of a fully dynamically-typed language.

Mixing Dependent Types and Simple Types. Several approaches have explored how soundly combine dependent types with non-dependently typed components. Ou et al. [2004] support a two-fragment language, where runtime checks at the boundary ensure that dependently-typed properties hold. The approach is limited to properties that are expressible as boolean-valued functions. Tanter and Tabareau [2015] develop a cast framework for subset types in Coq, allowing one to assert a term of type A to have the subset type $\{a : A \mid P\ a\}$ for any decidable (or checkable) property P. They use an axiom to represent cast errors. Osera et al. [2012] present *dependent interoperability* as a multi-language approach to combine both typing disciplines, mediated by runtime checks. Building on the subset cast framework (reformulated in a monadic setting instead of using axiomatic errors), Dagand et al. [2016, 2018] revisit dependent interoperability in Coq via type-theoretic Galois connections that allow automatic lifting of higher-order programs. Dependent interoperability allows exploiting the connection between pre-existing types, such as **Vec** and **List**, imposing the overhead of data structure conversions. The fact that **List** is a less precise structure than **Vec** is therefore defined *a posteriori*. In contrast, in GDTL, one can simply define **List** A as an alias for **Vec** A **?**, thereby avoiding the need for deep structural conversions.

The work of Lehmann and Tanter [2017] on gradual refinement types includes some form of dependency in types. Gradual refinement types range from simple types to logically-refined types, i.e. subset types where the refinement is drawn from an SMT-decidable logic. Imprecise logical formulaa in a function type can refer to arguments and variables in context. This kind of value dependency is less expressive than the dependent type system considered here. Furthermore, GDTL is the first gradual language to allow for **?** to be used in both term and type position, and to fully embed the untyped lambda calculus.

Programming with Holes. Finally, we observe that using **?** in place of proof terms in GDTL is related to the concept of *holes* in dependently-typed languages. Idris [Brady 2013] and Agda [Norell 2009] both allow typechecking of programs with typed holes. The main difference between **?** and holes in these languages is that applying a hole to a value results in a stuck term, while in GDTL, applying **?** to a value produces another **?**.

Recently, Omar et al. [2019] describe Hazelnut, a language and programming system with *typed holes* that fully supports evaluation in presence of holes, including reduction *around* holes. The approach is based on Contextual Modal Type Theory [Nanevski et al. 2008]. It would be interesting to study whether the dependently-typed version of CMTT [Pientka and Dunfield 2008] could be combined with the evaluation approach of Hazelnut, and the IDE support, in order to provide a rich programming experience with gradual dependent types.

10 CONCLUSION, LIMITATIONS AND PERSPECTIVES

GDTL represents a glimpse of the challenging and potentially large design space induced by combining dependent types and gradual typing. Specifically, this work proposes approximate normalization as a novel technique for designing gradual dependently-typed languages, in a way that ensures termination of normalization and naturally satisfies the gradual guarantees.

Currently, GDTL lacks a number of features required of a practical dependently-typed programming language. While we have addressed the most pressing issue of supporting inductive types in Section 8, the metatheory of this extension, in particular the proof of strong normalization, is future work. It might also be interesting to consider pattern matching as the primitive notion for eliminating inductives as in Agda, instead of elimination principles as in Coq; the equalities implied by dependent matches could be turned into runtime checks for gradually-typed values.

Additionally, a practical dependently-typed language should support *implicit arguments* and higher-order unification to automatically instantiate polymorphic function calls. Adding these would require a distinction between an implicit argument, which has a unique but unknown value, and the unknown term **?**, which could represent different values in different runs of the program. This difference is not unlike the relation between type inference and gradual typing explored in the literature [Garcia and Cimini 2015; Siek and Vachharajani 2008].

Performance should also be considered. While the evidence-based semantics of gradual languages is useful as a reference semantics, developing a space-efficient cast or coercion calculus would be desirable. In the dependently-typed setting, erasure of computationally-irrelevant arguments is also crucial [Brady et al. 2003]. The interaction between runtime checks due to gradual typing and erasure is an interesting topic to explore.

Finally, we observe that, by resorting to **?** during normalization as soon as a function of unknown type is applied, GDTL is very conservative and consequently imprecise statically. There are several ways to gain more precision during normalization, while preserving termination. An obvious and easy-to-implement solution is to assign a fixed budget of normalization steps when applying a **?**-typed function. A more expressive approach would be to exploit the *termination contracts* recently developed by Nguyen et al. [2018].

REFERENCES

Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. 2016. Gradual Type-and-Effect Systems. *Journal of Functional Programming* 26 (Sept. 2016), 19:1–19:69.

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development.* Springer-Verlag.

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

Edwin Brady, Conor McBride, and James McKinna. 2003. Inductive Families Need Not Store Their Indices. In *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers (Lecture Notes in Computer Science)*, Stefano Berardi, Mario Coppo, and Ferruccio Damiani (Eds.), Vol. 3085. Springer, 115–129. https://doi.org/10.1007/978-3-540-24849-1_8

Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. 2014. Combining Proofs and Programs in a Dependently Typed Language. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 33–45. https://doi.org/10.1145/2535838.2535883

Iliano Cervesato and Frank Pfenning. 2003. A Linear Spine Calculus. *Journal of Logic and Computation* 13, 5 (2003), 639–688. https://doi.org/10.1093/logcom/13.5.639 arXiv:http://logcom.oxfordjournals.org/content/13/5/639.full.pdf+html

Chiyan Chen and Hongwei Xi. 2005. Combining Programming with Theorem Proving. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP '05)*. ACM, New York, NY, USA, 66–77. https://doi.org/10.1145/1086365.1086375

Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* The MIT Press.

Thierry Coquand and Gérard Huet. 1988. The calculus of constructions. *Information and Computation* 76, 2 (1988), 95 – 120. https://doi.org/10.1016/0890-5401(88)90005-3

Pierre-Evariste Dagand, Nicolas Tabareau, and Éric Tanter. 2016. Partial Type Equivalences for Verified Dependent Interoperability. In *Proceedings of the 21st ACM SIGPLAN Conference on Functional Programming (ICFP 2016)*. ACM Press, Nara, Japan, 298–310.

Pierre-Évariste Dagand, Nicolas Tabareau, and Éric Tanter. 2018. Foundations of dependent interoperability. *Journal of Functional Programming* 28 (2018), e9. https://doi.org/10.1017/S0956796818000011

Nachum Dershowitz and Zohar Manna. 1979. Proving termination with multiset orderings. In *Automata, Languages and Programming*, Hermann A. Maurer (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 188–202.

Harley Eades and Aaron Stump. 2010. Hereditary substitution for stratified system F. In *International Workshop on Proof-Search in Type Theories, PSTT*, Vol. 10.

Luminous Fennell and Peter Thiemann. 2013. Gradual Security Typing with References. In *Proceedings of the 26th Computer Security Foundations Symposium (CSF)*. 224–239.

Ronald Garcia and Matteo Cimini. 2015. Principal Type Schemes for Gradual Programs. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 303–315. https://doi.org/10.1145/2676726.2676992

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 429–442. https://doi.org/10.1145/2837614.2837670

Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A Framework for Defining Logics. *J. ACM* 40, 1 (Jan. 1993), 143–184. https://doi.org/10.1145/138027.138060

Robert Harper and Daniel R. Licata. 2007. Mechanizing metatheory in a logical framework. *Journal of Functional Programming* 17, 4-5 (2007), 613–673. https://doi.org/10.1017/S0956796807006430

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

Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. 2008. AURA: a programming language for authorization and audit. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 27–38. https://doi.org/10.1145/1411204.1411212

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

Andres Löh, Conor McBride, and Wouter Swierstra. 2010. A Tutorial Implementation of a Dependently Typed Lambda Calculus. *Fundam. Inf.* 102, 2 (April 2010), 177–207. https://doi.org/10.3233/FI-2010-304

Cyprien Mangin and Matthieu Sozeau. 2015. Equations for Hereditary Substitution in Leivant's Predicative System F: A Case Study. *CoRR* abs/1508.00455 (2015). arXiv:1508.00455 http://arxiv.org/abs/1508.00455

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual Modal Type Theory. *ACM Trans. Comput. Logic* 9, 3, Article 23 (June 2008), 49 pages. https://doi.org/10.1145/1352582.1352591

Phuc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. 2018. Size-Change Termination as a Contract. arXiv:cs.PL/1808.02101

Ulf Norell. 2009. Dependently Typed Programming in Agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation (TLDI '09)*. ACM, New York, NY, USA, 1–2. https://doi.org/10.1145/1481861.1481862

Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live functional programming with typed holes. *PACMPL* 3, POPL (2019), 14:1–14:32. https://dl.acm.org/citation.cfm?id=3290327

Peter-Michael Osera, Vilhelm Sjöberg, and Steve Zdancewic. 2012. Dependent Interoperability. In *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification (PLPV '12)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/2103776.2103779

Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. 2004. Dynamic Typing with Dependent Types. In *Exploring New Frontiers of Theoretical Informatics*, Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell (Eds.). Springer US, Boston, MA, 437–450.

Frank Pfenning. 2008. Church and Curry: Combining intrinsic and extrinisic typing. In *Reasoning in Simple Type Theory – Festschrift in Honor of Peter B. Andrews on His 70th Birthday (Studies in Logic, Mathematical Logic and Foundations)*, Christoph Benzmüller, Chad Brown, Jörg Siekmann, and Richard Statman (Eds.). College Publications. http://www.collegepublications.co.uk/logic/mlf/?00010

Brigitte Pientka and Joshua Dunfield. 2008. Programming with Proofs and Explicit Contexts. In *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP '08)*. ACM, New York, NY, USA, 163–173. https://doi.org/10.1145/1389449.1389469

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 1–44. https://doi.org/10.1145/345099.345100

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

Tim Sheard and Nathan Linger. 2007. Programming in Omega. In *Central European Functional Programming School, Second Summer School, CEFP 2007, Cluj-Napoca, Romania, June 23-30, 2007, Revised Selected Lectures (Lecture Notes in Computer Science)*, Zoltán Horváth, Rinus Plasmeijer, Anna Soós, and Viktória Zsók (Eds.), Vol. 5161. Springer, 158–227. https://doi.org/10.1007/978-3-540-88059-2_5

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*. 81–92.

Jeremy G. Siek and Manish Vachharajani. 2008. Gradual typing with unification-based inference. In *Proceedings of the 4th ACM Dynamic Languages Symposium (DLS 2008)*. ACM Press, Paphos, Cyprus.

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

Vilhelm Sjöberg, Chris Casinghino, Ki Yung Ahn, Nathan Collins, Harley D. Eades III, Peng Fu, Garrin Kimmell, Tim Sheard, Aaron Stump, and Stephanie Weirich. 2012. Irrelevance, Heterogeneous Equality, and Call-by-value Dependent Type Systems. In Proceedings Fourth Workshop on *Mathematically Structured Functional Programming*, Tallinn, Estonia, 25 March 2012 *(Electronic Proceedings in Theoretical Computer Science)*, James Chapman and Paul Blain Levy (Eds.), Vol. 76. Open Publishing Association, 112–162. https://doi.org/10.4204/EPTCS.76.9

Aaron Stump. 2017. The calculus of dependent lambda eliminations. *Journal of Functional Programming* 27 (2017), e14. https://doi.org/10.1017/S0956796817000053

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 256–270. https://www.fstar-lang.org/papers/mumon/

Éric Tanter and Nicolas Tabareau. 2015. Gradual Certified Programming in Coq. In *Proceedings of the 11th Symposium on Dynamic Languages (DLS 2015)*. ACM, New York, NY, USA, 26–40. https://doi.org/10.1145/2816707.2816710

Matías Toro, Ronald Garcia, and Éric Tanter. 2018a. Type-Driven Gradual Security with References. *ACM Transactions on Programming Languages and Systems* 40, 4 (Nov. 2018), 16:1–16:55.

Matías Toro, Elizabeth Labrada, and Éric Tanter. 2018b. Gradual Parametricity, Revisited. arXiv:cs.PL/1807.04596

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. 2003. *A Concurrent Logical Framework I: Judgments and Properties*. Technical Report CMU-CS-02-101. https://www.cs.cmu.edu/~fp/papers/CMU-CS-02-101.pdf.

Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. ACM, New York, NY, USA, 214–227. https://doi.org/10.1145/292540.292560

A COMPLETE DEFINITIONS

A.1 SDTL

Static Terms

$$\begin{array}{lll} \mathsf{t},\ \mathsf{T} & ::= & \\ & \mid & \lambda x.\, \mathsf{t} \\ & \mid & \mathsf{t}_1\ \mathsf{t}_2 \\ & \mid & x \\ & \mid & (x : \mathsf{T}_1) \to \mathsf{T}_2 \\ & \mid & \mathbf{Set}_i \\ & \mid & \mathsf{t} :: \mathsf{T} \end{array}$$

Simple Contexts

$$\begin{array}{lll} C & ::= & \\ & \mid & \square\, \mathsf{t} \\ & \mid & \mathsf{v}\, \square \\ & \mid & (x : \square) \to \mathsf{T} \\ & \mid & \square :: \mathsf{T} \end{array}$$

Static Canonical Forms

$$\begin{array}{lll} \mathsf{u},\ \mathsf{U} & ::= & \\ & \mid & \lambda x.\, \mathsf{u} \\ & \mid & \mathsf{r} \\ & \mid & (x : \mathsf{U}_1) \to \mathsf{U}_2 \end{array}$$

Static Atomic Forms

$$\begin{array}{lll} \mathsf{r},\ \mathsf{R} & ::= & \\ & \mid & x\bar{\mathsf{s}} \\ & \mid & \mathbf{Set}_i \end{array}$$

Static Canonical Spines

$$\begin{array}{lll} \bar{\mathsf{s}} & ::= & \\ & \mid & \\ & \mid & \bar{\mathsf{s}}\ \mathsf{u} \end{array}$$

Static Environments

$$\begin{array}{lll} \Gamma & ::= & \\ & \mid & \cdot \\ & \mid & (x : \mathsf{U})\Gamma \\ & \mid & \Gamma_1\, \Gamma_2 \end{array}$$

$\boxed{\Gamma \vdash \mathsf{t} \Rightarrow \mathsf{U}}$ ***(Static Synthesis)***

$$\textsc{SSynthAnn}\quad \frac{\Gamma \vdash \mathsf{U} \leftsquigarrow \mathsf{T} : \mathbf{Set} \qquad \Gamma \vdash \mathsf{t} \Leftarrow \mathsf{U}}{\Gamma \vdash (\mathsf{t} :: \mathsf{T}) \Rightarrow \mathsf{U}}$$

$$\textsc{SSynthSet}\quad \frac{i > 0}{\Gamma \vdash \mathbf{Set}_i \Rightarrow \mathbf{Set}_{i+1}}$$

$$\textsc{SSynthVar}\quad \frac{\vdash \Gamma \qquad (x : \mathsf{U}) \in \Gamma}{\Gamma \vdash x \Rightarrow \mathsf{U}}$$

$$\textsc{SSynthApp}\quad \frac{\Gamma \vdash \mathsf{t}_1 \Rightarrow (x : \mathsf{U}_1) \to \mathsf{U}_2 \qquad \Gamma \vdash \mathsf{u} \leftsquigarrow \mathsf{t}_2 \Leftarrow \mathsf{U}_1 \qquad [\mathsf{u}/x]^{\mathsf{U}_1}\mathsf{U}_2 = \mathsf{U}_3}{\Gamma \vdash \mathsf{t}_1\ \mathsf{t}_2 \Rightarrow \mathsf{U}_3}$$

$\boxed{\Gamma \vdash \mathsf{t} \Leftarrow \mathsf{U}}$ ***(Static Checking)***

$$\textsc{SCheckSynth}\quad \frac{\Gamma \vdash \mathsf{t} \Rightarrow \mathsf{U}}{\Gamma \vdash \mathsf{t} \Leftarrow \mathsf{U}}$$

$$\textsc{SCheckLevel}\quad \frac{\Gamma \vdash \mathsf{T} \Rightarrow \mathbf{Set}_i \qquad 0 < i < j}{\Gamma \vdash \mathsf{T} \Leftarrow \mathbf{Set}_j}$$

$$\textsc{SCheckPi}\quad \frac{\Gamma \vdash \mathsf{U} \leftsquigarrow \mathsf{T}_1 \Leftarrow \mathbf{Set}_i \qquad \vdash (x : \mathsf{U})\Gamma \qquad (x : \mathsf{U})\Gamma \vdash \mathsf{T}_2 \Leftarrow \mathbf{Set}_i}{\Gamma \vdash (x : \mathsf{T}_1) \to \mathsf{T}_2 \Leftarrow \mathbf{Set}_i}$$

$$\textsc{SCheckLam}\quad \frac{\vdash (x : \mathsf{U}_1)\Gamma \qquad (x : \mathsf{U}_1)\Gamma \vdash \mathsf{t} \Leftarrow \mathsf{U}_2}{\Gamma \vdash (\lambda x.\, \mathsf{t}) \Leftarrow (x : \mathsf{U}_1) \to \mathsf{U}_2}$$

$\boxed{\Gamma \vdash t \rightsquigarrow u \Rightarrow U}$ ***(Static Normalization Synthesis)***

$$\textsc{SNormSynthAnn}\ \frac{\Gamma \vdash U \leftsquigarrow T : \mathbf{Set} \qquad \Gamma \vdash u \leftsquigarrow t \Leftarrow U}{\Gamma \vdash (t :: T) \rightsquigarrow u \Rightarrow U} \qquad \textsc{SNormSynthSet}\ \frac{i > 0}{\Gamma \vdash \mathbf{Set}_i \rightsquigarrow \mathbf{Set}_i \Rightarrow \mathbf{Set}_{i+1}} \qquad \textsc{SNormSynthVar}\ \frac{\vdash \Gamma \qquad (x : U) \in \Gamma \quad x \rightsquigarrow_\eta u : U}{\Gamma \vdash x \rightsquigarrow u \Rightarrow U}$$

$$\textsc{SNormSynthApp}\ \frac{\Gamma \vdash t_1 \rightsquigarrow (\lambda x.\, u_1) \Rightarrow (x : U_1) \to U_2 \qquad \Gamma \vdash u_2 \leftsquigarrow t_2 \Leftarrow U_1 \qquad [u_2/x]^{U_1} u_1 = u_3 \qquad [u_2/x]^{U_1} U_2 = U_3}{\Gamma \vdash t_1\ t_2 \rightsquigarrow u_3 \Rightarrow U_3}$$

$\boxed{\Gamma \vdash u \leftsquigarrow t \Leftarrow U}$ ***(Static Normalization Checking)***

$$\textsc{SNormCheckSynth}\ \frac{\Gamma \vdash t \rightsquigarrow u \Rightarrow U}{\Gamma \vdash u \leftsquigarrow t \Leftarrow U} \qquad \textsc{SNormCheckLevel}\ \frac{\Gamma \vdash T \rightsquigarrow U \Rightarrow \mathbf{Set}_i \qquad 0 < i < j}{\Gamma \vdash U \leftsquigarrow T \Leftarrow \mathbf{Set}_j}$$

$$\textsc{SNormCheckPi}\ \frac{\Gamma \vdash U_1 \leftsquigarrow T_1 \Leftarrow \mathbf{Set}_i \qquad \vdash (x : U_1)\Gamma \qquad (x : U_1)\Gamma \vdash U_2 \leftsquigarrow T_2 \Leftarrow \mathbf{Set}_i}{\Gamma \vdash (x : U_1) \to U_2 \leftsquigarrow (x : T_1) \to T_2 \Leftarrow \mathbf{Set}_i}$$

$$\textsc{SNormCheckLam}\ \frac{\vdash (x : U_1)\Gamma \qquad (x : U_1)\Gamma \vdash u \leftsquigarrow t \Leftarrow U_2}{\Gamma \vdash (\lambda x.\, u) \leftsquigarrow (\lambda x.\, t) \Leftarrow (x : U_1) \to U_2}$$

$\boxed{\Gamma \vdash U \leftsquigarrow T : \mathbf{Set}}$ ***(Normalization for Types with Unknown Level)***

$$\textsc{StaticSetNormSynth}\ \frac{\Gamma \vdash T \rightsquigarrow U \Rightarrow \mathbf{Set}_i}{\Gamma \vdash U \leftsquigarrow T : \mathbf{Set}} \qquad \textsc{StaticSetNormPi}\ \frac{\Gamma \vdash U_1 \leftsquigarrow T_1 : \mathbf{Set} \qquad (x : U_1)\Gamma \vdash U_2 \leftsquigarrow T_2 : \mathbf{Set}}{\Gamma \vdash (x : U_1) \to U_2 \leftsquigarrow (x : T_1) \to T_2 : \mathbf{Set}}$$

$\boxed{r \rightsquigarrow_\eta u : U}$ ***(Eta Expansion)***

$$\textsc{EtaExpandAtomic}\ \frac{}{r \rightsquigarrow_\eta r : R} \qquad \textsc{EtaExpandPi}\ \frac{y \rightsquigarrow_\eta u : U_1 \qquad x\overline{s}\ u \rightsquigarrow_\eta u : U_2}{x\overline{s} \rightsquigarrow_\eta (\lambda y.\, u) : (y : U_1) \to U_2}$$

$\boxed{\Gamma \vdash r \Rightarrow U}$ ***(Static Well-Formed Atomics)***

$$\textsc{SCSynthSet}\ \frac{i > 0}{\Gamma \vdash \mathbf{Set}_i \Rightarrow \mathbf{Set}_{i+1}} \qquad \textsc{SCSynthVar}\ \frac{\vdash \Gamma \qquad (x : U) \in \Gamma}{\Gamma \vdash x \Rightarrow U} \qquad \textsc{SCSynthApp}\ \frac{\Gamma \vdash x\overline{s} \Rightarrow (y : U_1) \to U_2 \qquad \Gamma \vdash u \Leftarrow U_1 \qquad [u/y]^{U_1} U_2 = U_3}{\Gamma \vdash x\overline{s}\ u \Rightarrow U_3}$$

$\boxed{\Gamma \vdash u \Leftarrow U}$ ***(Static Well-Formed Canonical Forms)***

SCCheckSynth
$$\frac{\Gamma \vdash r \Rightarrow R}{\Gamma \vdash r \Leftarrow R}$$

SCCheckLevel
$$\frac{\Gamma \vdash R \Rightarrow \mathbf{Set}_i \quad 0 < i < j}{\Gamma \vdash R \Leftarrow \mathbf{Set}_j}$$

SCCheckLam
$$\frac{\vdash (x : U_1)\Gamma \qquad (x : U_1)\Gamma \vdash u \Leftarrow U_2}{\Gamma \vdash (\lambda x.\, u) \Leftarrow (x : U_1) \to U_2}$$

SCCheckPi
$$\frac{\begin{array}{c}\Gamma \vdash U_1 \Leftarrow \mathbf{Set}_i \\ \vdash (x : U_1)\Gamma \qquad (x : U_1)\Gamma \vdash U_2 \Leftarrow \mathbf{Set}_i\end{array}}{\Gamma \vdash (x : U_1) \to U_2 \Leftarrow \mathbf{Set}_i}$$

$\boxed{\vdash \Gamma}$ ***(Well Formed Static Environments)***

SWFEmpty
$$\frac{}{\vdash \cdot}$$

SWFExt
$$\frac{\vdash \Gamma \qquad \Gamma \vdash U : \mathbf{Set} \qquad x \notin \Gamma}{\vdash (x : U)\Gamma}$$

$\boxed{\Gamma \vdash U : \mathbf{Set}}$ ***(Well-formed Types with Unknown Level)***

StaticSetSet
$$\frac{\Gamma \vdash r \Rightarrow \mathbf{Set}_i}{\Gamma \vdash r : \mathbf{Set}}$$

StaticSetPi
$$\frac{\begin{array}{c}\Gamma \vdash U_1 : \mathbf{Set} \quad \vdash (x : U_1)\Gamma \\ (x : U_1)\Gamma \vdash U_2 : \mathbf{Set}\end{array}}{\Gamma \vdash (x : U_1) \to U_2 : \mathbf{Set}}$$

$\boxed{[u_1/x]^{U} u_2 = u_3}$ ***(Static Hereditary Substitution)***

SHsubPi
$$\frac{\begin{array}{c}[u/x]^{U} U_1 = U_1' \\ [u/x]^{U} U_2 = U_2' \quad x \neq y\end{array}}{[u/x]^{U} (y : U_1) \to U_2 = (y : U_1') \to U_2'}$$

SHsubDiffNil
$$\frac{x \neq y}{[u/x]^{U} y = y}$$

SHsubDiffCons
$$\frac{\begin{array}{c}x \neq y \quad [u/x]^{U} y\bar{s} = y\bar{s}' \\ [u/x]^{U} u_2 = u_3\end{array}}{[u/x]^{U} y\bar{s}\ u_2 = y\bar{s}'\ u_3}$$

SHsubSet
$$\frac{}{[u/x]^{U} \mathbf{Set}_i = \mathbf{Set}_i}$$

SHsubLam
$$\frac{[u/x]^{U} u_2 = u_3 \quad x \neq y}{[u/x]^{U} (\lambda y.\, u_2) = (\lambda y.\, u_3)}$$

SHsubSpine
$$\frac{[u_1/x]^{U} x\bar{s}\ u_2 = u_3 \ : \ U'}{[u_1/x]^{U} x\bar{s}\ u_2 = u_3}$$

$\boxed{[u/x]^{U} x\bar{s} = u' \ : \ U'}$ ***(Static Atomic Hereditary Substitution)***

SHsubRHead
$$\frac{}{[u/x]^{U} x = u \ : \ U}$$

SHsubRSpine
$$\frac{\begin{array}{c}[u_1/x]^{U} x\bar{s} = (\lambda y.\, u_1') \ : \ (y : U_1') \to U_2' \qquad [u_1/x]^{U} u_2 = u_3 \\ [u_3/y]^{U_1'} u_1' = u_2' \qquad [u_3/y]^{U_1'} U_2' = U_3'\end{array}}{[u_1/x]^{U} x\bar{s}\ u_2 = u_2' \ : \ U_3'}$$

$\boxed{t_1 \longrightarrow t_2}$ ***(Simple Small-Step Semantics)***

SimpleStepAnn
$$\frac{}{(v :: T) \longrightarrow v}$$

SimpleStepApp
$$\frac{}{(\lambda x.\, t)\ v \longrightarrow [x \Mapsto v]t}$$

SimpleStepContext
$$\frac{t_1 \longrightarrow t_2}{C[t_1] \longrightarrow C[t_2]}$$

A.2 GDTL Surface Language

Gradual Terms

$$
\begin{array}{lcl}
t,\ T & ::= & \\
 & | & \lambda x.\, t \\
 & | & t_1\ t_2 \\
 & | & x \\
 & | & \mathbf{Set}_i \\
 & | & (x : T_1) \to T_2 \\
 & | & t :: T \\
 & | & ?
\end{array}
$$

Gradual Canonical Forms

$$
\begin{array}{lcl}
u,\ U & ::= & \\
 & | & \lambda x.\, u \\
 & | & r \\
 & | & ? \\
 & | & (x : U_1) \to U_2
\end{array}
$$

Gradual Canonical Spines

$$
\begin{array}{lcl}
\overline{s} & ::= & \\
 & | & \\
 & | & \overline{s}\ u
\end{array}
$$

Gradual Atomic Forms

$$
\begin{array}{lcl}
r,\ R & ::= & \\
 & | & x\overline{s} \\
 & | & \mathbf{Set}_i
\end{array}
$$

Gradual Environments

$$
\begin{array}{lcl}
\Gamma & ::= & \\
 & | & \cdot \\
 & | & (x : U)\Gamma \\
 & | & \Gamma_1\, \Gamma_2
\end{array}
$$

$\boxed{\Gamma \vdash t \Rightarrow U}$ ***(Gradual Synthesis)***

$$
\textsc{GSynthAnn}\ \frac{\Gamma \vdash U \leftsquigarrow T : \mathbf{Set} \qquad \Gamma \vdash t \Leftarrow U}{\Gamma \vdash (t :: T) \Rightarrow U}
\qquad
\textsc{GSynthSet}\ \frac{i > 0}{\Gamma \vdash \mathbf{Set}_i \Rightarrow \mathbf{Set}_{i+1}}
\qquad
\textsc{GSynthVar}\ \frac{(x : U) \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash x \Rightarrow U}
$$

$$
\textsc{GSynthApp}\ \frac{\Gamma \vdash t_1 \Rightarrow U \qquad \Gamma \vdash u \leftsquigarrow t_2 \Leftarrow \mathbf{dom}\ U \qquad [u/\_]\mathbf{cod}\ U = U_2}{\Gamma \vdash t_1\ t_2 \Rightarrow U_2}
\qquad
\textsc{GSynthDyn}\ \frac{}{\Gamma \vdash ? \Rightarrow ?}
$$

$\boxed{\Gamma \vdash t \Leftarrow U}$ ***(Gradual Checking)***

$$
\textsc{GCheckSynth}\ \frac{\Gamma \vdash t \Rightarrow U' \qquad U' \cong U}{\Gamma \vdash t \Leftarrow U}
\qquad
\textsc{GCheckLevel}\ \frac{\Gamma \vdash T \Rightarrow \mathbf{Set}_i \qquad 0 < i < j}{\Gamma \vdash T \Leftarrow \mathbf{Set}_j}
$$

$$
\textsc{GCheckPi}\ \frac{\Gamma \vdash U' \leftsquigarrow T_1 \Leftarrow U \qquad U \cong \mathbf{Set} \qquad \vdash (x : U')\Gamma \qquad (x : U')\Gamma \vdash T_2 \Leftarrow U}{\Gamma \vdash (x : T_1) \to T_2 \Leftarrow U}
$$

$$
\textsc{GCheckLamPi}\ \frac{\vdash (x : U_1)\Gamma \qquad (x : U_1)\Gamma \vdash t \Leftarrow U_2}{\Gamma \vdash (\lambda x.\, t) \Leftarrow (x : U_1) \to U_2}
\qquad
\textsc{GCheckLamDyn}\ \frac{\vdash (x : ?)\Gamma \qquad (x : ?)\Gamma \vdash t \Leftarrow ?}{\Gamma \vdash (\lambda x.\, t) \Leftarrow ?}
$$

$\boxed{\Gamma \vdash \mathrm{t} \rightsquigarrow \mathrm{u} \Rightarrow \mathrm{U}}$ ***(Approximate Normalization Synthesis)***

$$\textsc{GNSynthAnn}\quad \frac{\Gamma \vdash \mathrm{U} \leftsquigarrow \mathrm{T} : \mathbf{Set} \qquad \Gamma \vdash \mathrm{u} \leftsquigarrow \mathrm{t} \Leftarrow \mathrm{U}}{\Gamma \vdash (\mathrm{t} :: \mathrm{T}) \rightsquigarrow \mathrm{u} \Rightarrow \mathrm{U}}$$

$$\textsc{GNSynthSet}\quad \frac{i > 0}{\Gamma \vdash \mathbf{Set}_i \rightsquigarrow \mathbf{Set}_i \Rightarrow \mathbf{Set}_{i+1}}$$

$$\textsc{GNSynthVar}\quad \frac{\vdash \Gamma \qquad (x : \mathrm{U}) \in \Gamma \qquad x \rightsquigarrow_\eta \mathrm{u} : \mathrm{U}}{\Gamma \vdash x \rightsquigarrow \mathrm{u} \Rightarrow \mathrm{U}}$$

$$\textsc{GNSynthApp}\quad \frac{\begin{array}{c}\Gamma \vdash \mathrm{t}_1 \rightsquigarrow \mathrm{u}_1 \Rightarrow \mathrm{U} \qquad \mathrm{u}_1 \rightsquigarrow_\eta \mathrm{u}_1' : \mathbf{?} \to \mathbf{?} \\ \mathbf{dom}\ \mathrm{U} = \mathrm{U}_1 \qquad \Gamma \vdash \mathrm{u}_2 \leftsquigarrow \mathrm{t}_2 \Leftarrow \mathrm{U}_1 \\ [\mathrm{u}_2/\_]^{\mathrm{U}_1}\mathbf{body}\ \mathrm{u}' = \mathrm{u}_3 \\ [\mathrm{u}_2/\_]\mathbf{cod}\ \mathrm{U} = \mathrm{U}_2 \qquad \mathrm{u}_3 \rightsquigarrow_\eta \mathrm{u}_3' : \mathrm{U}_2\end{array}}{\Gamma \vdash \mathrm{t}_1\ \mathrm{t}_2 \rightsquigarrow \mathrm{u}_3' \Rightarrow \mathrm{U}_2}$$

$$\textsc{GNSynthDyn}\quad \frac{}{\Gamma \vdash \mathbf{?} \rightsquigarrow \mathbf{?} \Rightarrow \mathbf{?}}$$

$\boxed{\Gamma \vdash \mathrm{u} \leftsquigarrow \mathrm{t} \Leftarrow \mathrm{U}}$ ***(Approximate Normalization Checking)***

$$\textsc{GNCheckSynth}\quad \frac{\Gamma \vdash \mathrm{t} \rightsquigarrow \mathrm{u} \Rightarrow \mathrm{U}' \qquad \mathrm{U} \cong \mathrm{U}' \qquad \mathrm{U}' \sqsubseteq \mathrm{U}}{\Gamma \vdash \mathrm{u} \leftsquigarrow \mathrm{t} \Leftarrow \mathrm{U}}$$

$$\textsc{GNCheckApprox}\quad \frac{\Gamma \vdash \mathrm{t} \rightsquigarrow \mathrm{u} \Rightarrow \mathrm{U}' \qquad \mathrm{U} \cong \mathrm{U}' \qquad \mathrm{U}' \not\sqsubseteq \mathrm{U}}{\Gamma \vdash \mathbf{?} \leftsquigarrow \mathrm{t} \Leftarrow \mathrm{U}}$$

$$\textsc{GNCheckLevel}\quad \frac{\Gamma \vdash \mathrm{T} \rightsquigarrow \mathrm{U} \Rightarrow \mathbf{Set}_i \qquad 0 < i < j}{\Gamma \vdash \mathrm{U} \leftsquigarrow \mathrm{T} \Leftarrow \mathbf{Set}_j}$$

$$\textsc{GNCheckPi}\quad \frac{\begin{array}{c}\mathrm{U}_3 \cong \mathbf{Set} \\ \Gamma \vdash \mathrm{U}_1 \leftsquigarrow \mathrm{T}_1 \Leftarrow \mathrm{U}_3 \\ \vdash (x : \mathrm{U}_1)\Gamma \\ (x : \mathrm{U}_1)\Gamma \vdash \mathrm{U}_2 \leftsquigarrow \mathrm{T}_2 \Leftarrow \mathrm{U}_3\end{array}}{\Gamma \vdash (x : \mathrm{U}_1) \to \mathrm{U}_2 \leftsquigarrow (x : \mathrm{T}_1) \to \mathrm{T}_2 \Leftarrow \mathrm{U}_3}$$

$$\textsc{GNCheckLamPi}\quad \frac{\vdash (x : \mathrm{U}_1)\Gamma \qquad (x : \mathrm{U}_1)\Gamma \vdash \mathrm{u} \leftsquigarrow \mathrm{t} \Leftarrow \mathrm{U}_2}{\Gamma \vdash (\lambda x.\, \mathrm{u}) \leftsquigarrow (\lambda x.\, \mathrm{t}) \Leftarrow (x : \mathrm{U}_1) \to \mathrm{U}_2}$$

$$\textsc{GNCheckLamDyn}\quad \frac{\vdash (x : \mathbf{?})\Gamma \qquad (x : \mathbf{?})\Gamma \vdash \mathrm{u} \leftsquigarrow \mathrm{t} \Leftarrow \mathbf{?}}{\Gamma \vdash (\lambda x.\, \mathrm{u}) \leftsquigarrow (\lambda x.\, \mathrm{t}) \Leftarrow \mathbf{?}}$$

$\boxed{\Gamma \vdash \mathrm{U} \leftsquigarrow \mathrm{T} : \mathbf{Set}}$ ***(Approximate Normalization of Types with Unknown Level)***

$$\textsc{GradualSetNormSynth}\quad \frac{\Gamma \vdash \mathrm{T} \rightsquigarrow \mathrm{U} \Rightarrow \mathbf{Set}_i}{\Gamma \vdash \mathrm{U} \leftsquigarrow \mathrm{T} : \mathbf{Set}}$$

$$\textsc{GradualSetNormPi}\quad \frac{\Gamma \vdash \mathrm{U}_1 \leftsquigarrow \mathrm{SS} : \mathbf{Set} \qquad (x : \mathrm{U}_1)\Gamma \vdash \mathrm{U}_2 \leftsquigarrow \mathrm{T} : \mathbf{Set}}{\Gamma \vdash (x : \mathrm{U}_1) \to \mathrm{U}_2 \leftsquigarrow (x : \mathrm{SS}) \to \mathrm{T} : \mathbf{Set}}$$

$$\textsc{GradualSetNormDyn}\quad \frac{}{\Gamma \vdash \mathbf{?} \leftsquigarrow \mathbf{?} : \mathbf{Set}}$$

$\boxed{\mathrm{u} :_\eta \mathrm{U}}$ ***(Eta-long Canonical Forms)***

$$\textsc{GEtaLongAtomic}\quad \frac{}{\mathrm{r} :_\eta \mathrm{R}} \qquad \textsc{GEtaLongAtomicDyn}\quad \frac{}{\mathrm{r} :_\eta \mathbf{?}} \qquad \textsc{GEtaLongLam}\quad \frac{\mathrm{u} :_\eta \mathrm{U}}{(\lambda x.\, \mathrm{u}) :_\eta (x : \mathrm{U}') \to \mathrm{U}}$$

$$\textsc{GEtaLongDyn}\quad \frac{}{\mathbf{?} :_\eta \mathrm{U}} \qquad \textsc{GEtaLongPi}\quad \frac{}{(x : \mathrm{U}) \to \mathrm{U}' :_\eta \mathrm{U}''}$$

$\boxed{r \rightsquigarrow_\eta u : U}$ ***(Eta-expansion of Gradual Atomic Forms)***

$$\textsc{GEtaExpandAtomic}\ \frac{}{r \rightsquigarrow_\eta r : R} \qquad \textsc{GEtaExpandDyn}\ \frac{}{r \rightsquigarrow_\eta r : ?} \qquad \textsc{GEtaExpandPi}\ \frac{y \rightsquigarrow_\eta u : U_1 \quad x\overline{s}\ u \rightsquigarrow_\eta u_2 : U_2}{x\overline{s} \rightsquigarrow_\eta (\lambda y.\, u_2) : (y : U_1) \to U_2}$$

$\boxed{u \rightsquigarrow_\eta u' : U}$ ***(Eta-expansion of Gradual Canonical Forms)***

$$\textsc{GEtaExpandCAtomic}\ \frac{r \rightsquigarrow_\eta u : U}{r \rightsquigarrow_\eta u : U} \qquad \textsc{GEtaExpandCLam}\ \frac{u \rightsquigarrow_\eta u' : U_2}{(\lambda x.\, u) \rightsquigarrow_\eta (\lambda x.\, u') : (x : U_1) \to U_2}$$

$$\textsc{GEtaExpandCPi}\ \frac{U_1 \rightsquigarrow_\eta U_1' : U'' \quad U_2 \rightsquigarrow_\eta U_2' : U''}{(x : U_1) \to U_2 \rightsquigarrow_\eta (x : U_1') \to U_2' : U''} \qquad \textsc{GEtaExpandCDyn}\ \frac{}{? \rightsquigarrow_\eta ? : U} \qquad \textsc{GEtaExpandCDynType}\ \frac{}{u \rightsquigarrow_\eta u : ?}$$

$\boxed{\Gamma \vdash r \Rightarrow U}$ ***(Gradual Canonical Synthesis)***

$$\textsc{GCSynthSet}\ \frac{i > 0}{\Gamma \vdash \mathbf{Set}_i \Rightarrow \mathbf{Set}_{i+1}} \qquad \textsc{GCSynthVar}\ \frac{\vdash \Gamma \quad (x : U) \in \Gamma}{\Gamma \vdash x \Rightarrow U} \qquad \textsc{GCSynthApp}\ \frac{\Gamma \vdash x\overline{s} \Rightarrow U \quad \mathbf{dom}\ U = U_2 \quad \Gamma \vdash u \Leftarrow U_2 \quad [u/\_]\mathbf{cod}\ U = U_3}{\Gamma \vdash x\overline{s}\ u \Rightarrow U_3}$$

$\boxed{\Gamma \vdash u \Leftarrow U}$ ***(Gradual Canonical Checking)***

$$\textsc{GCCheckSynth}\ \frac{\Gamma \vdash r \Rightarrow U \quad U \cong U'}{\Gamma \vdash r \Leftarrow U'} \qquad \textsc{GCCheckLevel}\ \frac{\Gamma \vdash R \Rightarrow \mathbf{Set}_i \quad 0 < i < j}{\Gamma \vdash R \Leftarrow \mathbf{Set}_j} \qquad \textsc{GCCheckLamPi}\ \frac{\vdash (x : U_1)\Gamma \quad (x : U_1)\Gamma \vdash u \Leftarrow U_2}{\Gamma \vdash (\lambda x.\, u) \Leftarrow (x : U_1) \to U_2}$$

$$\textsc{GCCheckLamDyn}\ \frac{\vdash (x : ?)\Gamma \quad (x : ?)\Gamma \vdash u \Leftarrow ?}{\Gamma \vdash (\lambda x.\, u) \Leftarrow ?} \qquad \textsc{GCCheckPi}\ \frac{U_3 \cong \mathbf{Set} \quad \Gamma \vdash U_1 \Leftarrow U_3 \quad \vdash (x : U_1)\Gamma \quad (x : U_1)\Gamma \vdash U_2 \Leftarrow U_3}{\Gamma \vdash (x : U_1) \to U_2 \Leftarrow U_3} \qquad \textsc{GCCheckDyn}\ \frac{\Gamma \vdash U : \mathbf{Set}}{\Gamma \vdash ? \Leftarrow U}$$

$\boxed{\vdash \Gamma}$ ***(Well Formed Gradual Environments)***

$$\textsc{WFEmpty}\ \frac{}{\vdash \cdot} \qquad \textsc{WFExt}\ \frac{\vdash \Gamma \quad \Gamma \vdash U : \mathbf{Set} \quad x \notin \Gamma}{\vdash (x : U)\Gamma}$$

$\boxed{\Gamma \vdash U : \mathbf{Set}}$ ***(Well-Formed Gradual Types with Unkown Level)***

$$\textsc{GradualSetDynTy}\ \frac{\Gamma \vdash r \Rightarrow ?}{\Gamma \vdash r : \mathbf{Set}} \qquad \textsc{GradualSetSet}\ \frac{\Gamma \vdash r \Rightarrow \mathbf{Set}_i}{\Gamma \vdash r : \mathbf{Set}} \qquad \textsc{GradualSetPi}\ \frac{\Gamma \vdash U_1 : \mathbf{Set} \quad \vdash (x : U_1)\Gamma \quad (x : U_1)\Gamma \vdash U_2 : \mathbf{Set}}{\Gamma \vdash (x : U_1) \to U_2 : \mathbf{Set}} \qquad \textsc{GradualSetDynVal}\ \frac{}{\Gamma \vdash ? : \mathbf{Set}}$$

$\boxed{[u/x]^{U}\Gamma = \Gamma'}$ ***(Approximate Substitution on Gradual Environments)***

GradualEnvSubEmpty
$$\frac{}{[u/x]^{U}\cdot = \cdot}$$

GradualEnvSubVarDiff
$$\frac{x \neq y \quad [u/x]^{U}U = U' \quad [u/x]^{U}\Gamma = \Gamma'}{[u/x]^{U}(y : U)\Gamma = (y : U')\Gamma'}$$

GradualEnvSubVarSame
$$\frac{[u/x]^{U}\Gamma = \Gamma'}{[u/x]^{U}(x : U)\Gamma = \Gamma'}$$

$\boxed{[u_1/x]^{U}u_2 = u_3}$ ***(Approximate Hereditary Substitution)***

GHsubSet
$$\frac{}{[u/x]^{U}\mathbf{Set}_i = \mathbf{Set}_i}$$

GHsubDyn
$$\frac{}{[u/x]^{U}? = ?}$$

GHsubPi
$$\frac{x \neq y \quad [u/x]^{U}U_1 = U_1' \quad [u/x]^{U}U_2 = U_2'}{[u/x]^{U}(y : U_1) \to U_2 = (y : U_1') \to U_2'}$$

GHsubLam
$$\frac{x \neq y \quad [u/x]^{U}u_2 = u_3}{[u/x]^{U}(\lambda y.\, u_2) = (\lambda y.\, u_3)}$$

GHsubDiffNil
$$\frac{x \neq y}{[u/x]^{U}y = y}$$

GHsubDiffCons
$$\frac{x \neq y \quad [u/x]^{U}y\bar{s} = y\bar{s}' \quad [u/x]^{U}u_2 = u_3}{[u/x]^{U}y\bar{s}\ u_2 = y\bar{s}'\ u_3}$$

GHsubSpine
$$\frac{[u/x]^{U}x\bar{s} = u_2 : U_2}{[u/x]^{U}x\bar{s} = u_2}$$

$\boxed{[u/x]^{U}x\bar{s} = u_2 : U_2}$ ***(Approximate Atomic Hereditary Substitution)***

GHsubRHead
$$\frac{}{[u/x]^{U}x = u : U}$$

GHsubRDynSpine
$$\frac{[u/x]^{U}x\bar{s} = ? : (y : U_1) \to U_2 \qquad [u_2/y]^{U_1}U_2 = U_3}{[u/x]^{U}x\bar{s}\ u_2 = ? : U_3}$$

GHsubRLamSpine
$$\frac{\begin{array}{c}[u/x]^{U}x\bar{s} = (\lambda y.\, u_2) : (y : U_1) \to U_2 \qquad [u/x]^{U}u_1 = u_3 \\ [u_3/y]^{U_1}u_2 = u_4 \qquad [u_3/y]^{U_1}U_2 = U_3 \qquad u_4 \rightsquigarrow_{\eta} u_5 : U_3\end{array}}{[u/x]^{U}x\bar{s}\ u_1 = u_5 : U_3}$$

GHsubRDynType
$$\frac{[u_1/x]^{U}x\bar{s} = u_2 : ?}{[u_1/x]^{U}x\bar{s}\ u_2 = ? : ?}$$

$\boxed{U \cong U'}$ ***(Consistency of Gradual Canonical Terms)***

ConsistentEq
$$\frac{}{u \cong u}$$

ConsistentPi
$$\frac{U_1 \cong U_1' \quad U_2 \cong U_2'}{(x : U_1) \to U_2 \cong (x : U_1') \to U_2'}$$

ConsistentLam
$$\frac{u \cong u'}{(\lambda x.\, u) \cong (\lambda x.\, u')}$$

ConsistentApp
$$\frac{x\bar{s} \cong x\bar{s}' \quad u \cong u'}{x\bar{s}\ u \cong x\bar{s}'\ u'}$$

ConsistentDynL
$$\frac{}{? \cong u}$$

ConsistentDynR
$$\frac{}{u \cong ?}$$

$\boxed{U \cong \mathbf{Set}}$ ***(Set Consistency with Unknown Level)***

ConsistentSetSet
$$\frac{}{\mathbf{Set}_i \cong \mathbf{Set}}$$

ConsistentSetDyn
$$\frac{}{? \cong \mathbf{Set}}$$

$\boxed{U_1 \sqcap U_2 = U_3}$ *(Precision Meet of Gradual Canonical Forms)*

MeetPi
$$\frac{U_1 \sqcap U_1' = U_1'' \quad U_2 \sqcap U_2' = U_2''}{(x : U_1) \to U_2 \sqcap (x : U_1') \to U_2' = (x : U_1'') \to U_2''}$$

MeetLam
$$\frac{u \sqcap u' = u''}{\lambda x.\, u \sqcap \lambda x.\, u' = \lambda x.\, u''}$$

MeetApp
$$\frac{u \sqcap u' = u'' \quad x\overline{s} \sqcap x\overline{s}' = x\overline{s}''}{x\overline{s}\ u \sqcap x\overline{s}'\ u' = x\overline{s}''\ u''}$$

MeetDynL
$$\frac{}{? \sqcap U = U}$$

MeetDynR
$$\frac{U \neq ?}{U \sqcap ? = U}$$

MeetRefl
$$\frac{u \neq ?}{u \sqcap u = u}$$

$\boxed{U \sqsubseteq U'}$ *(Precision of Canonical Forms)*

MorePrecisePi
$$\frac{U_1 \sqsubseteq U_1' \quad U_2 \sqsubseteq U_2'}{(x : U_1) \to U_2 \sqsubseteq (x : U_1') \to U_2'}$$

MorePreciseLam
$$\frac{u \sqsubseteq u'}{\lambda x.\, u \sqsubseteq \lambda x.\, u'}$$

MorePreciseApp
$$\frac{u \sqsubseteq u' \quad x\overline{s} \sqsubseteq x\overline{s}'}{x\overline{s}\ u \sqsubseteq x\overline{s}'\ u'}$$

MorePreciseDyn
$$\frac{}{U \sqsubseteq ?}$$

MorePreciseRefl
$$\frac{U \neq ?}{U \sqsubseteq U}$$

$\boxed{\mathbf{dom}\ U_1 = U_2}$ *(Gradual Domain)*

DomainPi
$$\frac{}{\mathbf{dom}\ (x : U_1) \to U_2 = U_1}$$

DomainDyn
$$\frac{}{\mathbf{dom}\ ? = ?}$$

$\boxed{[u/\_]\mathbf{cod}\ U = U'}$ *(Gradual Codomain Substitution)*

CodSubPi
$$\frac{[u/x]^{U_1} U_2 = U_2'}{[u/\_]\mathbf{cod}\ (x : U_1) \to U_2 = U_2'}$$

CodSubDyn
$$\frac{}{[u/\_]\mathbf{cod}\ ? = ?}$$

$\boxed{[u/\_]^{U}\mathbf{body}\ u_2 = u_3}$ *(Gradual Function Body Substitution)*

BodySubPi
$$\frac{[u/x]^{U} u_2 = u_2'}{[u/\_]^{U}\mathbf{body}\ (\lambda x.\, u_2) = u_2'}$$

BodySubDyn
$$\frac{}{[u/\_]^{U}\mathbf{body}\ ? = ?}$$

A.3 GDTL Elaboration and Evidence Term Rules

Evidence Terms

$$e, E ::= \lambda x.\, e \mid e_1\ e_2 \mid x \mid (x : E_1) \to E_2 \mid \mathbf{Set}_i \mid \mathbf{?} \mid \varepsilon\, e \mid \mathsf{err}$$

Evidence Values

$$v, V ::= \varepsilon\, w \mid (x : V) \to E \mid \lambda x.\, e \mid \mathbf{Set}_i \mid \mathbf{?}$$

Raw Values

$$w, W ::= \lambda x.\, e \mid (x : V) \to E \mid \mathbf{Set}_i \mid \mathbf{?}$$

Evidence Contexts

$$C ::= \square\, e \mid v\, \square \mid (x : \square) \to E \mid \varepsilon\, \square$$

$\boxed{\Gamma \vdash u \leftsquigarrow e \Leftarrow U}$ ***(Additional Checking Rules for Evidence Term Normalization)***

GNCheckEvUp
$$\frac{\Gamma \vdash e \rightsquigarrow u \Rightarrow U' \qquad U \cong U' \quad U' \sqsubseteq U}{\Gamma \vdash u \leftsquigarrow \varepsilon\, e \Leftarrow U}$$

GNCheckEvDown
$$\frac{\Gamma \vdash e \rightsquigarrow u \Rightarrow U' \quad U' \not\sqsubseteq U}{\Gamma \vdash \mathbf{?} \leftsquigarrow \varepsilon\, e \Leftarrow U}$$

$\boxed{\varepsilon \vdash U \cong U'}$ ***(Consistency Supported by Evidence)***

EvConsistentDef
$$\frac{U_1 \sqcap U_2 = U_3 \quad U \sqsubseteq U_3}{\langle \Gamma, U \rangle \vdash U_1 \cong U_2}$$

$\boxed{\Gamma \vdash t \twoheadrightarrow e \Rightarrow U}$ ***(Gradual Synthesis Elaboration)***

GElabSynthAnn
$$\frac{\Gamma \vdash U \leftsquigarrow T : \mathbf{Set} \qquad \Gamma \vdash e \leftarrow t \Leftarrow U}{\Gamma \vdash (t :: T) \twoheadrightarrow e \Rightarrow U}$$

GElabSynthSet
$$\frac{i > 0}{\Gamma \vdash \mathbf{Set}_i \twoheadrightarrow \mathbf{Set}_i \Rightarrow \mathbf{Set}_{i+1}}$$

GElabSynthVar
$$\frac{\vdash \Gamma \quad (x : U) \in \Gamma}{\Gamma \vdash x \twoheadrightarrow x \Rightarrow U}$$

GElabSynthVarLook
$$\frac{}{\Gamma \vdash x \twoheadrightarrow e \Rightarrow U}$$

GElabSynthApp
$$\frac{\Gamma \vdash t_1 \twoheadrightarrow e_1 \Rightarrow U \qquad \mathbf{dom}\ U = U_1 \qquad \Gamma \vdash u \leftsquigarrow e_2 < \ t_2 \ <= U_1 \qquad [u/\_]\mathbf{cod}\ U = U_2}{\Gamma \vdash t_1\ t_2 \twoheadrightarrow e_1\ e_2 \Rightarrow U_2}$$

GElabSynthDyn
$$\frac{}{\Gamma \vdash \mathbf{?} \twoheadrightarrow \langle \Gamma, \mathbf{?} \rangle\, \mathbf{?} \Rightarrow \mathbf{?}}$$

$\boxed{\Gamma \vdash e \leftarrowtriangle t \Leftarrow U}$ ***(Gradual Checking Elaboration)***

$$\textsc{GElabCheckSynth}\ \frac{\Gamma \vdash t \rightarrowtriangle e \Rightarrow U' \qquad U \sqcap U' = U''}{\Gamma \vdash \langle \Gamma, U'' \rangle\, e \leftarrowtriangle t \Leftarrow U} \qquad \textsc{GElabCheckLevel}\ \frac{\Gamma \vdash T \rightarrowtriangle E \Rightarrow \mathbf{Set}_i \qquad 0 < i < j}{\Gamma \vdash E \leftarrowtriangle T \Leftarrow \mathbf{Set}_j}$$

$$\textsc{GElabCheckPi}\ \frac{U \cong \mathbf{Set} \qquad \Gamma \vdash U_1 \leftsquigarrow E_1 < \;\, T_1 <= U \qquad \vdash (x : U_1)\Gamma \qquad (x : U_1)\Gamma \vdash E_2 \leftarrowtriangle T_2 \Leftarrow U}{\Gamma \vdash (x : E_1) \longrightarrow E_2 \leftarrowtriangle (x : T_1) \longrightarrow T_2 \Leftarrow U}$$

$$\textsc{GElabCheckLamPi}\ \frac{\vdash (x : U_1)\Gamma \qquad (x : U_1)\Gamma \vdash e \leftarrowtriangle t \Leftarrow U_2}{\Gamma \vdash (\lambda x.\, e) \leftarrowtriangle (\lambda x.\, t) \Leftarrow (x : U_1) \longrightarrow U_2}$$

$$\textsc{GElabCheckLamDyn}\ \frac{\vdash (x : ?)\Gamma \qquad (x : ?)\Gamma \vdash e \leftarrowtriangle t \Leftarrow ?}{\Gamma \vdash (\lambda x.\, e) \leftarrowtriangle (\lambda x.\, t) \Leftarrow ?}$$

$\boxed{\Gamma \vdash e : U}$ ***(Evidence Term Typing)***

$$\textsc{EvTypeVar}\ \frac{\vdash \Gamma \qquad (x : U) \in \Gamma}{\Gamma \vdash x : U} \qquad \textsc{EvTypeApp}\ \frac{\Gamma \vdash e_1 : U \qquad \Gamma \vdash e_2 : \mathbf{dom}\ U \qquad [e_2/\_]\mathbf{cod}\ U = U_2}{\Gamma \vdash e_1\ e_2 : U_2}$$

$$\textsc{EvTypePi}\ \frac{U \cong \mathbf{Set} \qquad \Gamma \vdash U' \leftsquigarrow E_1 \Leftarrow U \qquad \vdash (x : U')\Gamma \qquad (x : U')\Gamma \vdash E_2 : U}{\Gamma \vdash (x : E_1) \longrightarrow E_2 : U}$$

$$\textsc{EvTypeSet}\ \frac{i > 0}{\Gamma \vdash \mathbf{Set}_i : \mathbf{Set}_{i+1}} \qquad \textsc{EvTypeEv}\ \frac{\Gamma \vdash e : U' \qquad \varepsilon \vdash U' \cong U}{\Gamma \vdash \varepsilon\, e : U} \qquad \textsc{EvTypeLevel}\ \frac{\Gamma \vdash E : \mathbf{Set}_i \qquad 0 < i < j}{\Gamma \vdash E : \mathbf{Set}_j}$$

$$\textsc{EvTypeLamPi}\ \frac{\vdash (x : U_1)\Gamma \qquad (x : U_1)\Gamma \vdash e : U_2}{\Gamma \vdash (\lambda x.\, e) : (x : U_1) \longrightarrow U_2} \qquad \textsc{EvTypeLamDyn}\ \frac{\vdash (x : ?)\Gamma \qquad (x : ?)\Gamma \vdash e : ?}{\Gamma \vdash (\lambda x.\, e) : ?}$$

$$\textsc{EvTypeDyn}\ \frac{\Gamma \vdash U : \mathbf{Set} \qquad \varepsilon \vdash U \cong U}{\Gamma \vdash \varepsilon\, ? : U}$$

$\boxed{[e/\_]\mathbf{cod}\ U_1 = U_2}$ ***(Codomain Subtitution with Evidence Terms)***

$$\textsc{EvCodSub}\ \frac{\cdot \vdash u \leftsquigarrow e \Leftarrow \mathbf{dom}\ U \qquad [u/\_]\mathbf{cod}\ U = U_2}{[e/\_]\mathbf{cod}\ U = U_2}$$

$\boxed{e_1 \longrightarrow e_2}$ ***(Evidence-based Small-Step Semantics)***

StepAscr
$$\frac{\varepsilon_1 \sqcap \varepsilon_2 = \varepsilon_3}{\varepsilon_1\,(\varepsilon_2\,w) \longrightarrow \varepsilon_3\,w}$$

StepAscrFail
$$\frac{\varepsilon_1 \sqcap \varepsilon_2\ \textbf{undefined}}{\varepsilon_1\,(\varepsilon_2\,w) \longrightarrow \mathsf{err}}$$

StepAppEv
$$\frac{\varepsilon_2 \sqcap \textbf{dom}\ \varepsilon_1 = \varepsilon_3 \qquad [w/\_]\textbf{cod}\ \varepsilon_1 = \varepsilon_4}{(\varepsilon_1\,(\lambda x.\,e))\,(\varepsilon_2\,w) \longrightarrow \varepsilon_4\,([x \Mapsto \varepsilon_3\,w]e)}$$

StepAppEvRaw
$$\frac{\textbf{dom}\ \varepsilon_1 = \varepsilon_2 \qquad [w/\_]\textbf{cod}\ \varepsilon_1 = \varepsilon_3}{(\varepsilon_1\,(\lambda x.\,e))\,w \longrightarrow \varepsilon_3\,([x \Mapsto \varepsilon_2\,w]e)}$$

StepApp
$$\frac{}{(\lambda x.\,e)\,v \longrightarrow [x \Mapsto v]e}$$

StepAppDyn
$$\frac{[v/\_]\textbf{cod}\ \varepsilon_1 = \varepsilon_2}{(\varepsilon_1\,?)\,v \longrightarrow \varepsilon_2\,?}$$

StepAppFailTrans
$$\frac{\textbf{dom}\ \varepsilon_1 = \varepsilon_3 \qquad \varepsilon_3 \sqcap \varepsilon_2\ \textbf{undefined}}{(\varepsilon_1\,w_1)\,(\varepsilon_2\,w_2) \longrightarrow \mathsf{err}}$$

StepAppFailDom
$$\frac{\textbf{dom}\ \varepsilon_1\ \textbf{undefined}}{(\varepsilon_1\,w)\,v \longrightarrow \mathsf{err}}$$

StepContext
$$\frac{e_1 \longrightarrow e_2 \qquad e_1, e_2 \neq \mathsf{err}}{C[e_1] \longrightarrow C[e_2]}$$

StepContextErr
$$\frac{e \longrightarrow \mathsf{err}}{C[e] \longrightarrow \mathsf{err}}$$

B PROOFS

We can define a function $\lceil t \rceil$ that maps $\lambda x.\,t$ to $(\lambda x.\,t) :: ?$.

Theorem 7.3. *For any term t, if Γ maps all variables to type ?, then $\Gamma \vdash \lceil t \rceil \Rightarrow ?$.*

Proof. We perform induction on $\lceil t \rceil$.

For a variable x, by our premise, its type in Γ is ?.

If $\lceil t \rceil = t'\ t''$, then by our premise t' and t'' both synthesize ?. This means that t'' checks against ? and ? checks against $\mathbf{Set}_i$ for any i. Finally, the codomain substitution of ? is ?, so $t'\ t''$ synthesizes ?.

If $\lceil t \rceil = (\lambda x.\,t') :: ?$, then ? is well-kinded with canonical form ?. We need to show that $(\lambda x.\,t')$ checks against ?. To do so, we apply the rule GCheckLamDyn. By our hypothesis, t' synthesizes ? in context $(x : ?)\Gamma$, so it must check against ?. □

The following formulation of type preservation for substitution is based on that of Pfenning [2008].

Lemma B.1 (Approximate Substitution Preserves Typing). *Suppose $\Gamma'\,(x : U)\Gamma \vdash y\bar{s} \Rightarrow U'$, $\Gamma \vdash u \Leftarrow U$. Let $[u/x]^U\Gamma' = \Gamma''$, $[u/x]^U x\bar{s} = u'' : U'''$, and $[u/x]^U U' = U''$.*

(1) If $x \neq y$, then $u'' = y\bar{s}'$ and $\Gamma''\,\Gamma \vdash y\bar{s}' \Rightarrow U''$, and $U'' = U'''$

(2) If $x = y$, then $\Gamma''\,\Gamma \vdash u'' \Leftarrow U''$

Similarly, suppose $\Gamma'\,(x : U)\Gamma \vdash u' \Leftarrow U'$, $\Gamma \vdash u \Leftarrow U$. Let $[u/x]^U\Gamma' = \Gamma''$, $[u/x]^U u' = u''$, and $[u/x]^U U' = U''$.

(3) Then $\Gamma''\,\Gamma \vdash u'' \Leftarrow U''$

Proof. By induction on the derivation of u''. For most rules, the typing derivation is trivial to construct using our inductive hypotheses, so we give the interesting rules here.

- GHsubDiffNil: Since y is unchanged by substitution, we must show that substitution on the environment produces the right type. If $(y : U') \in \Gamma$, then U' cannot contain x, so $(y : U') \in \Gamma$ and $[u/x]^U U' = U'$. If $(y : U') \in \Gamma'$, then $(y : U'') \in \Gamma''$ by the definition of substitution on environments. In both cases, y synthesizes the desired type.

- GHsubDiffCons: Suppose $\Gamma'(x : \mathrm{U})\Gamma \vdash y\overline{s}\ \mathrm{u}' \Rightarrow \mathrm{U}'$. Then $\Gamma'(x : \mathrm{U})\Gamma \vdash y\overline{s} \Rightarrow \mathrm{U}'_0$, the domain of U'_0 is defined, and $[\mathrm{u}'/\_]\mathbf{cod}\ \mathrm{U}'_0 = \mathrm{U}'$
If $\mathrm{U}'_0 = ?$, then $\mathbf{dom}\ \mathrm{U}'_0 = ?$, and $\Gamma'(x : \mathrm{U})\Gamma \vdash \mathrm{u}' \Leftarrow ?$, and $\mathrm{U}' = \mathrm{U}'' = ?$. If $\overline{s}', \mathrm{u}''$ are the substituted versions of $\overline{s}, \mathrm{u}'$ respectively, by our hypothesis we know that $\Gamma''\Gamma' \vdash y\overline{s}' \Rightarrow ?$ and $\Gamma''\Gamma' \vdash \mathrm{u}'' \Leftarrow ?$. By the definition of codomain, $[\mathrm{u}''/\_]\mathbf{cod}\ ? = ?$, which gives us enough to construct our desired typing derivation.
- GHsubRHead: In this case, $\mathrm{u}' = x$, $\mathrm{U}' = \mathrm{U}$ and $\mathrm{u}'' = \mathrm{u}$. By our premise, $\Gamma''(x : \mathrm{U})\Gamma \vdash x \Rightarrow \mathrm{U}$, with $\mathrm{U}' = \mathrm{U}$ and for the context to be well formed, this means that x cannot occur in U, so $\mathrm{U}'' = \mathrm{U}$. By our premise, combined with the fact that typing is preserved under context extension, $\Gamma''\Gamma \vdash \mathrm{u} \Leftarrow \mathrm{U}$.
- GHsubRLamSpine: In this case, $\mathrm{u}' = x\overline{s}\ \mathrm{u}'_1$, $[\mathrm{u}/x]^{\mathrm{U}_1} x\overline{s} = (\lambda y.\, \mathrm{u}_2) : (y : \mathrm{U}_1) \to \mathrm{U}_2$, and $[\mathrm{u}/x]^{\mathrm{U}_1}\mathrm{u}' = \mathrm{u}'_2$. By our hypothesis, $\Gamma''\Gamma \vdash \mathrm{u}'_2 \Leftarrow \mathrm{U}_2$ where $\Gamma''\Gamma \vdash (\lambda y.\, \mathrm{u}_2) \Leftarrow (y : \mathrm{U}_1) \to \mathrm{U}_2$. However, the function must be typed using GCCheckLam, meaning that $(y : \mathrm{U}_1)\Gamma''\Gamma \vdash \mathrm{u}_2 \Leftarrow \mathrm{U}_2$. We can once again apply our inductive hypothesis here to see that $\Gamma''\Gamma \vdash \mathrm{u}'_3 \Leftarrow \mathrm{U}'$, proving our result.
- GHsubRDynType or GHsubDynSpine: The result holds since ? checks against any type that is itself well-typed. ? checks against any $\mathbf{Set}_i$. For GHsubDynSpine, our hypothesis says that $\Gamma''\Gamma' \vdash ? \Leftarrow (y : \mathrm{U}'_1) \to \mathrm{U}'_2$, so $\Gamma''\Gamma' \vdash (y : \mathrm{U}'_1) \to \mathrm{U}'_2 \Leftarrow \mathbf{Set}_i$ for some i. This in turn means that $(y : \mathrm{U}'_1)\Gamma''\Gamma' \vdash \mathrm{U}'_2 \Leftarrow \mathbf{Set}_i$. We can again use our hypothesis to show that $\Gamma''\Gamma \vdash \mathrm{V3} \Leftarrow \mathbf{Set}_i$.

Note that for the last two cases, the induction is well-founded: since Γ'' and Γ are not part of the derivation of the substitution, we can quantify them universally in our inductive hypothesis. □

Theorem 5.2 (Normalization preserves typing). *If* $\Gamma \vdash \mathrm{u} \rightsquigarrow \mathrm{t} \Leftarrow \mathrm{U}$, *then* $\Gamma \vdash \mathrm{u} \Leftarrow \mathrm{U}$.

Proof. We perform mutual induction, proving that if $\Gamma \vdash \mathrm{t} \rightsquigarrow \mathrm{u} \Rightarrow \mathrm{U}$, then $Gamma \vdash gu <= gU$. All cases are simple, except for the following:
GNCheckSynth: since $\mathrm{U}' \sqsubseteq \mathrm{U}$, the static gradual guarantee gives us that t checks against U.
GNCheckApprox: ? checks against any type.
GNSynthApp: while we only know that the normal form of t_1 checks against the type that t_1 synthesizes, this is enough to fulfill the premises of preservation of typing by approximate substitution, which, combined with our hypothesis, gives us our result. □

Lemma B.2. *Suppose* $\Gamma \vdash \mathrm{u} \Leftarrow \mathrm{U}$ *and* $\Gamma'(x : \mathrm{U})\Gamma \vdash \mathrm{u}' \Leftarrow \mathrm{U}'$. *Then there exists some* u'' *such that* $[\mathrm{u}/x]^{\mathrm{U}}\mathrm{u}' = \mathrm{u}''$.
Similarly, if $\Gamma \vdash \mathrm{u} \Leftarrow \mathrm{U}$ *and* $\Gamma'(x : \mathrm{U})\Gamma \vdash \mathrm{r} \Rightarrow \mathrm{U}'$. *Then there exists some* u'' *such that* $[\mathrm{u}/x]^{\mathrm{U}}\mathrm{r} = \mathrm{u}''$, *and if* $\mathrm{r} = x\overline{s}\ \mathrm{u}'$, $[\mathrm{u}/x]^{\mathrm{U}} x\overline{s}\ \mathrm{u}' = \mathrm{u}'' : \mathrm{U}'$.

Proof. We perform nested induction: first, on the multiset of universes of arrow-types, then on the typing derivation for u'.

- GCSynthSet, GCCheckDyn: take $\mathrm{u}'' = \mathrm{u}'$.
- GCSyntVar: take $\mathrm{u}'' = y$ if $x \neq y$, u otherwise.
- GCCheckLamDyn, GCCheckLamPi, GCCheckPi, GCCheckSynth: follows immediately from our hypothesis.
- GCSynthApp: This case is the most interesting. Suppose we are substituting into $x\overline{s}\ \mathrm{u}'$. If $x \neq y$, then $y\overline{s}$ and u' must both be well typed, so we can apply our hypothesis to find their substituted forms.

Assume then that $x = y$. By our premise, there must be some type such that $\Gamma'(x : U)\Gamma \vdash (x\overline{s}) \Rightarrow U_3$, and it must have a defined domain.
If $U_3 = ?$, then by our hypothesis, there's some value where $[u/x]^{U} x\overline{s} = u'' : ?$, meaning we can apply GHsubRDynType to produce $? : ?$ as our result. Note that if $U = ?$, we necessarily have this case.
Otherwise, for the domain to be defined, $U_3 = (y : U_1) \to U_2$. By our hypothesis, there must be some value such that $[u/x]^{U} x\overline{s} = u'' : U''$, and moreover, preservation of typing says that $[u/x]^{U}(y : U_1) \to (U_2) = U''$ and that $\Gamma''\Gamma \vdash u'' \Leftarrow U''$, where Γ'' is Γ' with u substituted for x. But this means that $U'' = (y : U_1') \to U_2'$, since we must have constructed it using GHsubPi. Then, if U'' an arrow type, it is not atomic, meaning that u'' could only check against it using GCCheckDyn or GCCheckLamPi.
In both cases, since the whole application is well typed, we know that $\Gamma'(x : U)\Gamma \vdash u' \Leftarrow U_1$, so there's some value where $[u/x]^{U} u' = u_2'$. Preservation of typing gives that $\Gamma''\Gamma \vdash u_2' \Leftarrow U_1'$. Also, U_1' is less than U in our multiset order.
In the first case, $u'' = ?$. Then we can use $?$ as our return value, and by our outer inductive hypothesis, there must be some value such that $[u_2'/y]^{U_1'} U_2 = U'$, giving us our return type. This allows us to build the derivation with GHsubDynSpine.
In the second case, $u'' = (\lambda y.\, u_2)$. We can decompose the typing of u'' to see that $(y : U_1')\Gamma''\Gamma \vdash u_2 \Leftarrow U_2'$, which lets us apply our outer inductive hypothesis see that $[u_2'/y]^{U_1'} u_2 = u_3'$, which we can use as our final result. Similarly, $[u_2'/y]^{U_1'} U_2' = U_3'$ gives us our final type, which allows us to construct the derivation using GHsubRLamSpine.

□

The fact that the normalization rules directly correspond to typing rules, combined with the totality of hereditary substitution, is enough to give us our result for normalization.

Theorem 5.3 (Normalization is Total). *If $\Gamma \vdash t \Leftarrow U$, then $\Gamma \vdash u \rightsquigarrow t \Leftarrow U$ for exactly one u.*

Lemma B.3 (Progress). *If $\cdot \vdash e' : U$ and e' is not a value, then there exists some e such that $e' \longrightarrow e$.*

Proof. By induction on the derivation on the typing of e'.

- EvTypeSet, EvTypeLevel, EvTypeDyn, EvTypeLamPi, EvTypeLamDyn: must be values
- EvTypeEv,EvTypePi: either we have a value, or can step with our context rules.
- EvTypeVar: cannot be typed under empty environment
- EvTypeAppDep: then $e' = e''\ e$. If e'' or e is not a value, then we can step by the context rule. Otherwise, $\cdot \vdash e'' : (x : U) \to U'$. By inversion on our typing rules, If e'' is a function, we can step with StepApp. If e'' is $\langle U_1 \rangle\, w$, either U_1 has a defined domain and codomain, in which case we apply StepAppDyn, or it does not, in which case we apply StepAppDyn-Fail. Otherwise, e'' is an evidence annotated function. Assume that e is annotated with evidence. If the evidence is a dependent function type, and e does not normalize, then we fail with StepAppFailCod. Otherwise, we can apply one of StepAppEv, StepAppFailTrans, or StepAppFailDom depending on whether the domain of the evidence is defined, and whether the meet with the evidence of e is defined. Finally, if e has no evidence, we repeat the above process after applying StepAppEvRaw.
- EvTypeAppSImple: as above, but since the function ignores its argument, we know that we will never have failure from normalizing the argument.

□

Lemma B.4 (Preservation). *If* $\cdot \vdash \mathrm{e}' : \mathrm{U}$ *and* $\mathrm{e}' \longrightarrow \mathrm{e}$, *then* $\mathrm{e} = \mathrm{err}$ *or* $\cdot \vdash \mathrm{e} : \mathrm{U}$.

Proof. By induction on the derivation of $\mathrm{e}' \longrightarrow \mathrm{e}$.

- StepAscrFail,StepAppFailTrans,StepAppFailDom,StepAppDynFail,StepContextErr: trivial, since we step to err.
- StepAscr: then $\mathrm{e}' = \langle \mathrm{U}_1 \rangle (\langle \mathrm{U}_2 \rangle \, \mathrm{e}'')$, and by inversion on typing, we know that $\cdot \vdash \mathrm{e}'' : \mathrm{U}'$ where $\langle \mathrm{U}_2 \rangle \vdash \mathrm{U}' \cong \mathrm{U}$. Since $\mathrm{U}_1 \sqcap \mathrm{U}_2 \sqsubseteq \mathrm{U}_2$ then $\langle \mathrm{U}_1 \sqcap \mathrm{U}_2 \rangle \vdash \mathrm{U}' \cong \mathrm{U}$, which gives us $\cdot \vdash \langle \mathrm{U}_1 \sqcap \mathrm{U}_2 \rangle \, \mathrm{e}'' : \mathrm{U}$.
- StepApp: this follows easily from the preservation of typing by approximate substitution.
- StepAppEv: then $\mathrm{e}' = (\langle \mathrm{U}_1 \rangle (\lambda x.\, \mathrm{e}''))\, (\langle \mathrm{U}_2 \rangle \, \mathrm{w})$. By inversion on typing, $\cdot \vdash \lambda x.\, \mathrm{e}'' : \mathrm{U}_1'$ where $\langle \mathrm{U}_1 \rangle \vdash \mathrm{U}_1' \cong \mathrm{U}_2'$, and the whole expression e' types against the codomain of U_2'. We know that $\Gamma \vdash \langle \mathrm{U}_2 \rangle \, \mathrm{w} : \mathbf{dom}\ \mathrm{U}_2'$, so $\langle \mathrm{U}_2 \rangle \vdash \mathbf{dom}\ \mathrm{U}_2' \cong \mathrm{U}_3$ and $\cdot \vdash \mathrm{w} : \mathrm{U}_3$.
  We then know that $\langle \mathrm{U}_2 \sqcap \mathbf{dom}\ \mathrm{U}_1 \rangle \vdash \mathbf{dom}\ \mathrm{U}_1' \cong \mathrm{U}_2'$. So $\cdot \vdash \langle \mathrm{U}_2 \sqcap \mathbf{dom}\ \mathrm{U}_1 \rangle \, \mathrm{w} : \mathbf{dom}\ \mathrm{U}_1'$. By preservation of typing under substitution, substituting this into e'' has type $[\mathrm{w}/\_]\mathbf{cod}\ \mathrm{U}_1'$. Finally, we then know that if $\langle [\mathrm{w}/\_]\mathbf{cod}\ \mathrm{U}_1 \rangle \vdash [\mathrm{w}/\_]\mathbf{cod}\ \mathrm{U}_1' \cong [\mathrm{w}/\_]\mathbf{cod}\ \mathrm{U}_2'$. This means that our final result can be typed at U_2'.
- StepAppDyn: holds by preservation of typing under codomain substitution.
- StepAppEvRaw: similar reasoning as for StepAppEv, except with less indirection since we know $\cdot \vdash \mathrm{w} : \mathbf{dom}\ \mathrm{U}_2'$.
- StepContext: holds by our inductive hypothesis and preservation under substitution (since hole-filling is a special case of substitution).

□

These together give us type soundness.

Theorem 7.1 (Type safety). *If* $\cdot \vdash \mathrm{e} : \mathrm{U}$, *then either* $\mathrm{e} \longrightarrow^* \mathrm{v}$ *for some* v, $\mathrm{e} \longrightarrow^* \mathrm{err}$, *or* e *diverges.*

We now show that our AGT functions form a Galois-connection.

Theorem B.5 (Soundness of α). *If* $A \neq \emptyset$, *then* $A \subseteq \gamma(\alpha(A))$.

Proof. By induction on the structure of $\alpha(A)$.

If $\alpha(A) = x\bar{s}'\ \mathrm{u}'$ then $A = \{\, x\bar{s}\ \mathrm{u} \mid x\bar{s} \in B_1, \mathrm{u} \in B_2 \,\}$ for some B_1, B_2 where $\alpha(B_1) = x\bar{s}', \alpha(B_2) = \mathrm{u}'$. Then $\gamma(x\bar{s}'\ \mathrm{u}') = \{\, x\bar{s}'\ \mathrm{u}' \mid x\bar{s}' \in \gamma(x\bar{s}'), \mathrm{u}' \in \gamma(\mathrm{u}') \,\}$. By our hypothesis, $B_1 \subseteq \gamma(x\bar{s}')$ and $B_2 \subseteq \gamma(\mathrm{u}')$, so $\{\, x\bar{s}\ \mathrm{u} \mid x\bar{s} \in B_1, \mathrm{u} \in B_2 \,\} \subseteq \{\, x\bar{s}'\ \mathrm{u}' \mid x\bar{s}' \in \gamma(x\bar{s}'), \mathrm{u}' \in \gamma(\mathrm{u}') \,\}$. The cases for λ and $\longrightarrow$ are proved in the same way.

If $\alpha(A) = x$, then $A = \{\, x \,\}$ and $\gamma(\alpha(A)) = \{\, x \,\}$. The same logic proves the case for **Set**.

If $\alpha(A) = ?$, $\gamma(\alpha(A)) = \mathrm{SCanonical} \supseteq A$ by definition. □

Theorem B.6 (Optimality of α). *If* $A \neq \emptyset$, $A \subseteq \gamma(\mathrm{u})$ *then* $\alpha(A) \sqsubseteq \mathrm{u}$.

Proof. By induction on the structure of u.

Case $\mathbf{Set}_i$: then $\gamma(\mathrm{u}) = \{\, \mathbf{Set}_i \,\}$, so if A is nonempty $A = \{\, \mathbf{Set}_i \,\}$. The same reasoning holds if $\mathrm{u} = x$.

Case $x\bar{s}\ \mathrm{u}$: then $\gamma(\mathrm{u}) = \{\, xe'u' \mid x\bar{s}' \in \gamma(x\bar{s}), \mathrm{u}' \in \gamma(\mathrm{u}) \,\}$. If A is nonempty and $A \subseteq \gamma(\mathrm{u})$, then there are some set B_1, B_2 such that $B_1 \subseteq \gamma(x\bar{s})$, $B_2 \subseteq \gamma(\mathrm{u})$, and $A = \{\, x\bar{s}_2\ \mathrm{u}_2 \mid x\bar{s}_2 \in B_1, \mathrm{u}_2 \in B_2 \,\}$. So $\alpha(A) = x\bar{s}_3\ \mathrm{u}_3$ where $\alpha(B_1) = x\bar{s}_3$, $\alpha(B_2) = \mathrm{u}_3$. Neither B_1 or B_2 can be empty if A is non-empty, so by our inductive hypothesis, $x\bar{s}_3 \sqsubseteq x\bar{s}$ and $\mathrm{u}_3 \sqsubseteq \mathrm{u}$. So $x\bar{s}_3\ \mathrm{u}_3 \sqsubseteq x\bar{s}\ \mathrm{u}$, giving us our result. The same reasoning holds for the λ and $\longrightarrow$ cases.

Case ?: we have our result, since $\mathrm{u}' \sqsubseteq ?$ holds for all gu'. □

Lemma B.7. *Suppose* $u_1 \sqsubseteq^\eta u_1'$, $u_2 \sqsubseteq^\eta u_2'$ *and* $U \sqsubseteq^\eta U'$ *and* $U_2 \sqsubseteq^\eta U_2'$, *where* $\Gamma \vdash u_2 \Leftarrow U_2$ *and* $\Gamma' \vdash u_2' \Leftarrow U_2'$ *for some* $\Gamma \sqsubseteq^\eta \Gamma'$. *If* $[u_1/x]^U u_2 = u_3$ *and* $[u_1'/x]^{U'} u_2' = u_3'$, *then* $u_3 \sqsubseteq^\eta u_3'$.

Suppose also that $x\overline{s} \sqsubseteq^\eta x\overline{s}'$, *where* $[u_1/x]^U x\overline{s} = u_3 : U_3$, $[u_1'/x]^{U'} x\overline{s}' = u_3' : U_3'$, $\Gamma \vdash x\overline{s} \Rightarrow U_2$ *and* $\Gamma' \vdash x\overline{s}' \Rightarrow U_2'$. *Then* $u_3 \sqsubseteq^\eta u_3'$ *and* $U_3 \sqsubseteq^\eta U_3'$.

Proof. First we note that if $u_2' = ?$, then $u_3' = ?$, and the result is trivially true.

We then assume that $u_2' \neq ?$, and proceed by induction on the derivation of $[u_1/x]^U u_2 = u_3$.

GHsubSet, GHsubDiffNil: In these cases, $u_2 =^\eta u_2' =^\eta u_3 =^\eta u_3'$.

GHsubPi, GHsubLam, : follows from the inductive hypothesis.

GHsubLam: This is the case in which we must consider η-equality. Here, $u_2 = \lambda y.\, u_4$. If $u_2' = \lambda y.\, u_4'$, then the result follows from our inductive hypothesis. Otherwise, it must be some $z\overline{s}'$ that η-expands to a (possibly) less-precise version of u_2. Since both terms are well-formed, they are η-long with respect to U_2 and U_2' respectively. So U_2 must be ? or an arrow-type, but U_2' cannot be an arrow type, so $U_2 = ?$. Then we know that $u_4 \sqsubseteq^\eta z\overline{s}'\ y$, so by our hypothesis, if $[u_1/x]^U u_4 = u_5$ and $[u_1'/x]^{U'} z\overline{s}'\ y = u_5'$ then $u_5' \sqsubseteq^\eta u_5'$. We know that $x \neq y$ by our premise. If $x \neq z$, then we know that $u_5' = z\overline{s}'\ y$, so $\lambda y.\, u_5' \sqsubseteq^\eta z\overline{s}'$. If $x = z$, then $[u_1'/x]^{U'} z\overline{s}'$ has type ?, so applying it to y produces ?. Then $u_5' = ?$, giving us our result.

GHsubSpine: Then $u_1 = x\overline{s}$. Since $U_2 \sqsubseteq^\eta U_2'$, u_2' cannot possibly be η-expanded any further than $x\overline{s}$m so it must be equal to some $x\overline{s}'$. The result then follows from our hypothesis.

GHsubDiffCons: same logic as the previous case.

GHsubRHead: Then $u_2 = u_2' = x$, $u_3 = u_1$ and $u_3' = u_1'$. Likewise, $U = U_2$ and $U' = U_2'$, so our result follows from our premises. The only other case is where u_2' is an η-expansion of x.

GHsubRDynSpine: then $u_1 = x\overline{s}\ u_4$ and $u_1' = x\overline{s}'\ u_4'$, where $x\overline{s} \sqsubseteq^\eta x\overline{s}'$ and $u_4 \sqsubseteq^\eta u_4'$. By our premise, $[u_1/x]^U x\overline{s} = ? : U_2$, so by our hypothesis, $[u_1'/x]^{U'} x\overline{s}' = ? : U_2'$, so $\overline{s}_3' = ?$, giving us our result.

GHsubRLamSpine: follows from applying our inductive hypothesis to each sub-derivation.

GHsubRDynType: then $u_1 = x\overline{s}\ u_4$ and $u_1' = x\overline{s}'\ u_4'$, where $x\overline{s} \sqsubseteq^\eta x\overline{s}'$ and $u_4 \sqsubseteq^\eta u_4'$. By our premise, $[u_1/x]^U x\overline{s} = u_4 : ?$, so by our hypothesis, $[u_1'/x]^{U'} x\overline{s}' = u_4' : ?$, since ? is the least precise type. Then $u_3 = u_3' = ?$.

□

Corollary B.8. *If* $u_1 \sqsubseteq^\eta u_1'$, *and* $U \sqsubseteq^\eta U'$, $[u_1/\_]\mathbf{cod}\ U = U_2$ *and* $[u_1'/\_]\mathbf{cod}\ U' = U_2'$, *then* $U_2 \sqsubseteq^\eta U_2'$.

Corollary B.9. *If* $u_1 \sqsubseteq^\eta u_1'$, $u_2 \sqsubseteq^\eta u_2'$, $U \sqsubseteq^\eta U'$, $[u_1/\_]^U \mathbf{body}\ u_2 = u_3$ *and* $[u_1'/\_]^{U'} \mathbf{body}\ u_2' = u_3'$, *then* $u_3 \sqsubseteq^\eta u_3'$.

Lemma 7.5 (Normalization Gradual Guarantee). *Suppose* $\Gamma_1 \vdash u_1 \leftsquigarrow t_1 \Leftarrow U_1$. *If* $\Gamma_1 \sqsubseteq^\eta \Gamma_2$, $t_1 \sqsubseteq t_2$, *and* $U_1 \sqsubseteq^\eta U_2$, *then* $\Gamma_2 \vdash u_2 \leftsquigarrow t_2 \Leftarrow U_2$ *where* $u_1 \sqsubseteq^\eta u_2$.

Proof. We first note that if $t_2 = ?$, then $u_2 = ?$, so $u_1 \sqsubseteq u_2$.

Assume then that $t_2 \neq ?$. We proceed by induction on the derivation of the normalization of t_1. We perform mutual induction to prove the same result for synthesis.

GNSynthAnn: follows immediately from inductive hypothesis.

GNSynthSet: trivial

GNSynthVar: trivial, since we consider precision modulo η-expansion

GNSynthApp: follows from our inductive hypothesis, Corollary B.9, Corollary B.8, and the transitivity of precision.

GNCheckSynth,GNCheckLevel, GNCheckPi, GNCheckLamPi, GNCheckLamDyn, GNCheck-Dyn: follows immediately from our inductive hypothesis.

GNCheckSynth: we note that decreasing the precision of a term can only decrease the precision of its synthesized type. Given this, we know that if t synthesizes U', then t_2 synthesizes U'', where also $U'' \not\sqsubseteq U$. So both normalize to ?.

GNCheckDyn: vacuous, t_2 must be ? in this case. ◻

We note that we can apply the exact same proof procedure to achieve the same result for elaborated terms.

Lemma B.10. *If* $v \sqsubseteq e$*, then* e *is a syntactic value.*

Proof. If $e = ?$, then it is a value. Otherwise we proceed by induction on the structure of v, noting that a less precise value must have the same top-level constructor if it is not ?. For $\varepsilon\, w$ and $(x : w) \to E$, it follows by applying our hypothesis to w and W respectively. $\mathbf{Set}_i$ and ? are trivial, and a function is a value regardless of its body, giving us our result. ◻

Theorem 7.4 (Gradual Guarantee).
(Static Guarantee) Suppose $\Gamma \vdash t \Leftarrow U$ *and* $U \sqsubseteq U'$*. If* $\Gamma \sqsubseteq \Gamma'$ *and* $t \sqsubseteq t'$*, then* $\Gamma_2 \vdash t' \Leftarrow U'$*.*
(Dynamic Guarantee) Suppose that $\cdot \vdash e_1 : U$, $\cdot \vdash e_1' : U'$, $e_1 \sqsubseteq e_1'$*, and* $U \sqsubseteq U'$*. Then if* $e_1 \longrightarrow^* e_2$*, then* $e_1' \longrightarrow^* e_2'$ *where* $e_2 \sqsubseteq e_2'$*.*

Proof of Static Guarantee. We prove by mutual induction with the following proposition: if $\Gamma \vdash t \Rightarrow U$, $\Gamma \sqsubseteq \Gamma'$ and $t \sqsubseteq t'$, then $\Gamma' \vdash t' \Rightarrow U'$ for some U' where $U \sqsubseteq U'$.

First we note that if $t' = ?$, then our results holds trivially: ? synthesizes the least precise type, and can check against any type using GCheckSynth. We hence assume that $t' \neq ?$, and proceed by induction on the typing derivation for t.

GSynthAnn: then $t = t_1 :: T$, and $t' = t_1' :: T'$. By our premise, T has some normal form U_1. By Lemma 7.5, T' has a normal form U_1' where $U_1 \sqsubseteq U_1'$. By our inductive hypothesis, $\Gamma \vdash t_1' \Leftarrow U_1'$, allowing us to complete our typing derivation.

GSynthSet: then $t = t' = \mathbf{Set}_i$ and we can use an identical typing derivation.

GSynthVar: then $t = t' = x$, and by our premise that $\Gamma \sqsubseteq \Gamma'$, the synthesized type U from Γ is at least as precise as U' from Γ'.

GSynthApp: then $t = t_1\ t_2$, and $t' = t_1'\ t_2'$. The result then follows from our hypothesis, combined with the monotonicity of domain and codomain.

GSynthDyn: vacuous.

GCheckSynth: our hypothesis gives that $\Gamma \vdash t \Rightarrow U_1$ and $\Gamma' \vdash t' \Rightarrow U_1'$ where $U_1 \sqsubseteq U_1'$. Since we know $U \sqsubseteq U'$, we know that $U_1' \cong U'$, giving us our typing derivation.

GCheckLevel: if $\Gamma \vdash t \Rightarrow \mathbf{Set}_i$, by our hypothesis, either $\Gamma' \vdash t' \Rightarrow \mathbf{Set}_i$, or $\Gamma' \vdash t' \Rightarrow ?$. In the first case, we can type t' with GCheckLevel, and in the second we can use GCheckSynth.

GCheckPi: follows from our hypothesis and Lemma 7.5.

GCheckLamPi, GCheckLamDyn: follows from our hypothesis. ◻

Proof of Dynamic Guarantee. We prove a slightly stronger result: If $e_1 \longrightarrow e_2$, where $e_2 \neq \mathrm{err}$ and $e_1 \sqsubseteq e_1'$, then $e_1' \longrightarrow^* e_2'$ for some e_2' such that $e_2 \sqsubseteq e_2'$.

We first note that if $e_1' = ?$, then the result trivially holds since $? \longrightarrow^* ?$. We assume then that $e_1' \neq ?$, and proceed by induction on the derivation of $e_1 \longrightarrow e_2$.

StepAscrFail, StepAppFailTrans, StepAppFailDom, StepContextErr: vacuous.

StepAscr: then $e_1 = \varepsilon_1 (\varepsilon_2\, w)$ and $e_1' = ep1'(ep2'rv')$. If $e_1 \sqsubseteq e_1'$, then $\varepsilon_1 \sqsubseteq \varepsilon_1'$, $\varepsilon_2 \sqsubseteq \varepsilon_2'$ and $w_1 \sqsubseteq w_1'$. Our premise gives that $\varepsilon_1 \sqcap \varepsilon_2$ is defined, and since we define meet and precision on sets of static values, $\varepsilon_1 \sqcap \varepsilon_2 \sqsubseteq \varepsilon_1' \sqcap \varepsilon_2'$. So $e_1 \longrightarrow \langle \varepsilon_1' \sqcap \varepsilon_2' \rangle\, w'$ and $\langle \varepsilon_1 \sqcap \varepsilon_2 \rangle\, w \sqsubseteq \langle \varepsilon_1' \sqcap \varepsilon_2' \rangle\, w'$.

StepAppEv: Then $e_1 = (\varepsilon_1 (\lambda x.\, e_3))\, (\varepsilon_2\, w)$. We have two cases for our precision relation to hold.

In the first, $e_2 = (\varepsilon_1' \, ?)\, (\varepsilon_2'\, w')$ where $\varepsilon_1 \sqsubseteq \varepsilon_1'$ and $\varepsilon_2 \sqsubseteq \varepsilon_2'$. By our premise, $[\varepsilon_2\, w/\_]\mathbf{cod}\ \varepsilon_1 = \varepsilon_4$ for some ε_4, so there must be some ε_4' where $[\varepsilon_2'\, w'/\_]\mathbf{cod}\ \varepsilon_1' = \varepsilon_4'$, and by Corollary B.8, $\varepsilon_4 \sqsubseteq \varepsilon_4'$. So we can step $(\varepsilon_1' \, ?)\, (\varepsilon_2'\, w') \longrightarrow \varepsilon_4' \, ?$ by StepAppDyn, and since $\varepsilon_4 \sqsubseteq \varepsilon_4'$ and $?$ is less precise than all terms, we have our result.

In the second case, $e_1' = (\varepsilon_1 (\lambda x.\, e_3'))\, (\varepsilon_2'\, w)$. Then we can step e_1' using StepAppEv. By definition of **dom**, Corollary B.8 and the monotonicity of the precision meet, the evidences created to step e_1' are all no more precise than the corresponding ones for e_1. Since syntactic substitution preserves precision, we have our result.

StepAppEvRaw: By the same argument as StepAppEv, except that in the second case we need not apply monotonicity of the meet.

StepApp: holds because syntactic substitution preserves precision.

StepAppDyn: Then $e = (\varepsilon_1 \, ?)\, v$, so for precision to hold, e_1' must be $(\varepsilon_1' \, ?)\, v'$, where $\varepsilon_1 \sqsubseteq \varepsilon_1'$ and $v \sqsubseteq v'$. By Corollary B.8, if $[v/\_]\mathbf{cod}\ \varepsilon_1 = \varepsilon_2$ and $[v/\_]\mathbf{cod}\ \varepsilon_1' = \varepsilon_2'$, then $\varepsilon_2 \sqsubseteq \varepsilon_2'$, so we can step $e_1' \longrightarrow \varepsilon_2' \, ?$ and our precision result holds.

StepContext: If $e_1 = C[e_3]$ and $e_1' = C[e_3']$, by our premise and inductive hypothesis, we have $e_3 \longrightarrow e_4$, $e_3' \longrightarrow e_4'$ and $e_4 \sqsubseteq e_4'$. If $e_1 = e_3\, e_5$, then $e_1' = e_3'\, e_5'$ and $e_5 \sqsubseteq e_5'$, so $e_4\, e_5 \sqsubseteq e_4'\, e_5'$, and we can step to $C[e_4']$ using StepContext, and preserve the precision relation. Similar reasoning shows the same result for the other possible frames, though we note that the fact that precision preserves the value property is required for the frames involving values. □