Propositional Equality for Gradual Dependently Typed Programming*

JOSEPH EREMONDI, University of British Columbia, Canada
RONALD GARCIA, University of British Columbia, Canada
ÉRIC TANTER, PLEIAD Lab, Computer Science Department (DCC), University of Chile, Chile

Gradual dependent types can help with the incremental adoption of dependently typed code by providing a principled semantics for *imprecise* types and proofs, where some parts have been omitted. Current theories of gradual dependent types, though, lack a central feature of type theory: propositional equality. Lennon-Bertrand et al. show that, when the reflexive proof *refl* is the only closed value of an equality type, a gradual extension of the Calculus of Inductive Constructions (CIC) with propositional equality violates static observational equivalences. Extensionally-equal functions should be indistinguishable at run time, but they can be distinguished using a combination of equality and type imprecision.

This work presents a gradual dependently typed language that supports propositional equality. We avoid the above issues by devising an equality type of which *refl* is not the only closed inhabitant. Instead, each equality proof is accompanied by a term that is at least as precise as the equated terms, acting as a witness of their plausible equality. These witnesses track partial type information as a program runs, raising errors when that information shows that two equated terms are undeniably inconsistent. Composition of type information is internalized as a construct of the language, and is deferred for function bodies whose evaluation is blocked by variables. We thus ensure that extensionally-equal functions compose without error, thereby preventing contexts from distinguishing them. We describe the challenges of designing consistency and precision relations for this system, along with solutions to these challenges. Finally, we prove important metatheory: type safety, conservative embedding of CIC, weak canonicity, and the gradual guarantees of Siek et al., which ensure that reducing a program's precision introduces no new static or dynamic errors.

CCS Concepts: • **Theory of computation** → **Type structures**; **Program semantics**.

Additional Key Words and Phrases: dependent types, gradual types, propositional equality

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

1 INTRODUCTION

Gradual dependent types relax the discipline of dependent types, so that programmers can write, type check and run programs with partial type information and omit yet-to-be-devised terms or proofs. These capabilities have the potential to help migrate code from non-dependently typed languages, and reduce the learning curve for newcomers to this rich but complex type discipline.

*This work is partially funded by CONICYT FONDECYT Regular Project 1190058. This work is partially funded by an NSERC Discovery Grant.

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, rxg@cs.ubc.ca; Éric Tanter, PLEIAD Lab, Computer Science Department (DCC), University of Chile, Santiago, Chile, etanter@dcc.uchile.cl.

2475-1421/2022/8-ART96
https://doi.org/10.1145/3547627

Gradual languages [Siek and Taha 2006] check programs against the type information statically available, comparing types via *consistency* $\cong$, i.e., equality up to missing type information. Static checks skipped due to partial type information are instead performed at run time when the actual values are known. Programs missing type information might not fail, nor are errors deferred until an unsafe operation is attempted. Rather, partial type information is exploited at run time, so an error occurs when a computation's result has a type incompatible with the context in which it arises. Gradual dependent types let programmers use type-driven programming with holes [Brady 2017] while running code and executing tests, even when missing parts of types, terms or proofs.

However, existing gradual dependent languages do not support *propositional equality* [Martin-Löf 1982]. The propositional equality type $t_1 =_T t_2$ expresses that t_1 and t_2 are equal inhabitants of type T. Its only constructor is $\mathsf{refl}_t : t =_T t$, the proof that every term is equal to itself. Equality is useful for practical dependently typed programming, since it lets a function express pre- and post-conditions by taking or returning equality proofs. Likewise, a programmer can use an equality proof to rewrite the type of the expression they are trying to produce. Propositional equality even lays a path to support GADT-style inductive families, since constructors with different return types can be encoded with non-indexed inductive types and propositional equality [McBride 2000].

Limited means of representing and reasoning about equality have been used in existing gradual languages. GCIC [Lennon-Bertrand et al. 2022] supports *decidable equality* (see §2.2), where a type is computed by pattern-matching on the equated terms. Gradual Refinement Types [Lehmann and Tanter 2017] support first-order constraints in linear integer arithmetic. In contrast, propositional equality is general and lightweight: it works for every type, provides its own construction and elimination principles, and can be used with quantifiers or higher order functions.

Until now, the challenge with gradual equality has been propagating and enforcing equality constraints at run time. The problem is that equated terms may contain functions or dependent function types, both of which bind variables. For example, $(\lambda x.\, x + 0) =_{\mathbb{N}} (\lambda x.\, x)$ and $((x : \mathbb{N}) \to \mathsf{Vec}\ \mathbb{N}\ (x + 0)) =_{\mathbf{Type}} ((x : \mathbb{N}) \to \mathsf{Vec}\ \mathbb{N}\ x)$ are well-formed types. Extensionally equality of functions (up to partial information) is undecidable. Comparing functions syntactically, by directly comparing bound variables, is decidable. Such a notion works for compile-time consistency checks, but would be problematic during run-time checks, since it destroys static reasoning principles. Observationally equivalent terms in the static language would be distinguishable in the gradual language, e.g., replacing $\lambda x.\, x + x$ with $\lambda x.\, 2 * x$ can cause new dynamic errors. Lennon-Bertrand et al. [2022] show that when *refl* is the only static constructor for propositional equality, its inclusion in a gradual language violates static equivalences. Moreover, code that compares bound variables cannot be easily compiled, since every function now needs a syntactic representation.

This paper presents the language GEq (pronounced "geek"), which adds propositional equality to GCIC, allowing $=$, *refl* and **J** to be used like in the Calculus of Inductive Constructions (CIC), but with a dynamic semantics that is meaningful for gradual types. Our **key insight** is to **represent an equality proof using a *witness* that captures equality constraints discovered at run time**. Taking inspiration from evidence in Abstracting Gradual Typing (AGT) [Garcia et al. 2016] and middle-types in threesomes [Siek and Wadler 2010], we represent witnesses with a term that is as precise as both equated terms. As a consequence, *refl* and **?** are *not* the only inhabitants of the equality type, avoiding the above impossibility result. Our **contributions** are as follows:

- We demonstrate how equality proofs between imprecise terms are useful for discovering bugs in programs and for guiding the development of static proofs (§2);
- We extend GCIC with propositional equality (§4) by typing equality using consistency witnesses between terms (§4.2). We give operational semantics via a cast calculus, where the eliminator for equality uses casts going through the result type given by the witness (§4.3). To combine witnesses

when casting between equality types, we add witness composition directly as a construct in GEq (§4.4). This operator delays the comparison of neutral terms until their variables are bound to values, so composing statically-equivalent functions does not raise an error;
- We prove type safety, conservative extension of CIC, weak canonicity, and the gradual guarantees for GEq (§5), so imprecision never causes stuck states or new (static or dynamic) errors, and GEq rejects ill-typed CIC programs. Like Siek and Chen [2021], our proofs are parameterized over definitions of consistency and precision, revealing sufficient properties to prove the theorems;
- We define precision and consistency for the cast calculus (§6), showing that they fulfill the previously-identified properties. We separate *static consistency*, whether a term of some type can be used in a given context, from *dynamic consistency*, whether two terms compose without error. These coincide for non-dependent gradual languages, but in GEq they must be separated to respect static equivalences while still rejecting ill-typed static programs.

Section 3 reviews GCIC and its cast calculus CastCIC, upon which the other sections build. Section 7 discusses extensions enabled by GEq's features, along with related and future work.

2 SETTING THE STAGE

2.1 Programming vs. Proving and the Gradual Guarantees

Though programming and proving are connected by the Curry-Howard correspondence, the language features best supporting each task differ. Our focus is dependently typed programming: we consider GEq as a model of a programming language rather than as a type theory for mechanizing mathematics. Nevertheless, we prove important metatheory about GEq that may aid in the development of future gradual type theories.

One goal with GEq is proving the *gradual guarantees* of Siek et al. [2015b], which state that a reduction in precision introduces no new static or dynamic errors. These guarantees are useful for programming because of the contrapositive: if a program has a type error, adding more type information does not remove the error. The types are fundamentally inconsistent and must be changed. By contrast, in current implementations of holes either block reduction, causing errors, or block type checking, hiding errors that would otherwise be statically detectable.

2.2 Relationship to Existing Languages

GEq builds off the work of two existing gradual dependent languages, GDTL [Eremondi et al. 2019] and GCIC [Lennon-Bertrand et al. 2022]. Section 7.3 gives a broader discussion of related work.

GDTL is a Gradual Dependently Typed Language with dependent functions, a universe hierarchy, and decidable type checking. It introduced the *imprecise term* **?**, which extended gradual typing to allow imprecision not only in types, but type indices and proof terms. GEq inherits **?** from GDTL. Since it is based on AGT [Bañados Schwerter et al. 2021; Garcia et al. 2016], GDTL features ideas similar to GEq's witnesses. However, the authors only discuss equality and inductive types as an extension, omitting it from their metatheory. Also, GDTL uses naive syntactic composition, and suffers from the issues we discuss in the introduction: fully static terms that are observationally-equivalent in the static language may have different run-time behavior in the gradual language.

GCIC is a Gradual version of the Calculus of Inductive Constructions (CIC). It uses a cast calculus approach, extending a restricted version of CIC with inductive types but no indexed inductive families or propositional equality. GEq is a direct extension of GCIC. The GCIC authors prove that no gradual language can simultaneously conservatively extend CIC, have strong normalization, and have *graduality*, a strengthening of the gradual guarantees where decreasing then increasing precision produces an equivalent term. The authors give three variants of GCIC, called $\text{GCIC}^{\mathcal{G}}$, $\text{GCIC}^{\mathcal{N}}$ and $\text{GCIC}^{\uparrow}$, which respectively sacrifice one of strong normalization, graduality, and

conservative extension of CIC, while keeping the other two properties. We build off $\text{GCIC}^{\mathcal{G}}$, because, of the three options, sacrificing strong normalization is most palatable for programming. $\text{GCIC}^{\mathcal{N}}$ violates the gradual guarantees, and $\text{GCIC}^{\uparrow}$ is too restrictive for practical programming, so we avoid them both. Logical inconsistency and non-terminating proofs are not as detrimental in programming as in mechanized mathematics: type safety is still guaranteed, and errors due to non-terminating proofs are likely to be discovered when a program is run.

GCIC has no dedicated equality type, but decidable equality types are supported. That is, programmers can write a function that takes two elements of a type, and produces a type that is inhabited if and only if they are equal. The programmer must construct, either manually or with tactics, an equality function for each type whose terms they wish to equate, along with the corresponding elimination principle. Most function types have undecidable equality, and hence are unsupported by this method. Also, common functions on equalities cannot be expressed in their most general form with this method, such as such as $\mathsf{cong} : (f : T \to S) \to a =_T b \to f\ a =_S f\ b$. Full propositional equality is more convenient for the programmer.

2.3 A Motivating Example: Eagerly Enforcing Specifications

In this section, we motivate our development with examples of how gradual dependent types can catch errors related to the lengths of lists. A guiding principle of our work is that the types the programmer writes should, as much as possible, be treated as specifications to be checked, either statically or dynamically, regardless of whether their enforcement is required for safety.

Throughout the paper, we write static terms using red sans-serif font. Terms from the gradual surface language use *green, italic serif font*. The theory is developed using a gradual cast calculus, which we write using **blue, bold serif font**.

A Buggy Quicksort: We begin by showing how gradual types help the migration of a sorting function from a non-dependently typed language to one with dependent types, and how this migration can help identify bugs. Consider a flawed quicksort implementation:

```
sort : List Float → List Float                 |  List A = Nil | Cons A (List A)
sort Nil = Nil                                 |  (++) : List A → List A → List A (concatenation)
sort (Cons h t) =(sort (filter (<  h) t)) ++  [h] ++ (sort (filter (>  h) t))
```

Since $<$ is used instead of $\leq$, duplicates are erroneously removed from the list. The programmer may have a suspicion that they have made a mistake in their code, or may have observed incorrect behavior while testing. Their dependent type enthusiast friends have repeatedly assured them that dependently typed languages can help eliminate bugs, so they try migrating their code to a dependently typed language with propositional equality.

An approach to reasoning about the correctness of sort is to use *Fixed-length Lists*, which we call FLists. Dependent pairs and propositional equality allow for a type of lists indexed by their length.[1] Figure 1 shows this type, and how it can be used to express that sort should preserve the length of the produced list. Here, refl is the reflexive proof that t is equal to itself. That sort function behaves like the non-dependent version, except it must extract the Lists from the FLists produced by the recursive calls, and produce an FList with proof that the length is the same as the input.

At this stage, the programmer must fill the hole ??? by constructing a proof of type $\mathsf{length}\ (\mathsf{Cons}\ h\ t) =_{\mathbb{N}} \mathsf{length}\ (\mathsf{sortLt} +\!\!+ [h] +\!\!+ \mathsf{sortGt})$. This task is difficult for a newcomer, since they must use associativity of addition and how $\mathsf{length}\ (\mathsf{list}_1 +\!\!+ \mathsf{list}_2) =_{\mathbb{N}} \mathsf{length}\ \mathsf{list}_1 + \mathsf{length}\ \mathsf{list}_2$.

[1] A more conventional approach would be to use an indexed type family, which we discuss in §7.1.

Moreover, they must prove that length t =ℕ (length lt) + (length gt), but such a proof is impossible, due to the bug. Even if they had such a proof, they would need to then use the proofs from the recursive calls, plt and pgt, to relate the lengths of lt and gt to the lengths of sortLt and sortGt.

The programmer is now cursing their type-theorist friend. For a non-buggy quicksort, one could construct the necessary proof, but doing so is difficult, particularly for a newcomer. The type checker does not detect the bug, so it does not inform the programmer that hole cannot be filled, and it cannot say which aspect of the proof is impossible. Also, development has now stopped: the programmer cannot run or test their code without the missing proof.

```
FList : Type → ℕ → Type
FList A n = ((x : List A) × length x =ℕ n)
sort : (n : ℕ) → FList Float n → FList Float n
sort 0 (Nil, p) = (Nil, p)
sort (1 + n) (Cons h t, p) =
    let lt = (filter (< h) t)
        (sortLt, plt) = sort (length lt) (lt, refl)
        gt = (filter (> h) t)
        (sortGt, pgt) = sort (length gt) (gt, refl)
    in (sortLt ++ [h] ++ sortGt, ???)
```

Fig. 1. Sorting Fixed-Length Lists

Gradual Types to the Rescue: GEq lets the programmer run and test *sort* before writing the missing proof, checking (within the limits of decidability) whether any static values could possibly replace imprecise types and proofs. In Fig. 1, replacing the hole with **?**, the *imprecise term,*[2] yields a complete, well-typed GEq function that can be called, tested, or used in other modules.

The utility of gradual typing is shown in the run-time checks that let us identify bugs in code. The run-time semantics of GEq are defined via type-directed elaboration to a cast calculus CastEq, in which all implicit conversions are replaced by explicit casts. During type checking, **?** is elaborated into the CastEq's $?_{\mathbf{n}=_{\mathbb{N}}\mathbf{length\ (sortLt +\!\!+ [h] +\!\!+ sortGt)}}$ (the least precise term of type $\mathbf{n} =_{\mathbb{N}} \mathbf{length\ (sortLt +\!\!+ [h] +\!\!+ sortGt)}$), which is not a value in CastEq. Instead, it reduces to the consistency witness $\mathbf{n} \,\&_{\mathbb{N}}\, \mathbf{length\ (sortLt +\!\!+ [h] +\!\!+ sortGt)}$. The operator & is the gradual composition operator, which combines information statically known about its operands. Because types depend on terms, composition is not limited to types, but can combine terms of any type. The & operator is a syntactic construct of CastEq, not a meta-operation like it is in existing literature [Siek et al. 2015a; Siek and Wadler 2010]. Reifying & into the object language is critical for composing functions (§2.4). Since **n** is a variable, this composition expression does not reduce further.

We can identify the bug once *sort* is applied to a concrete list. Consider the input [2.2, 1.1, 3.3, 2.2], which elaborates to [2.2, 1.1, 3.3, 2.2] in CastEq. Applying **sort** binds **lt** := [1.1] and **gt** := [3.3], giving a result list of [1.1, 2.2, 3.3]. Then **n** is 4 and **length (sortLt ++ [h] ++ sortGt)** is 3, so the witness for the result is the composition $4 \,\&_{\mathbb{N}}\, 3$, which reduces to a run-time error.

In a language without dependent types, this bug could be caught with testing or assertions. In GEq, however, dependent types provide a unified means of specifying properties to be checked statically or dynamically. During development, types serve as assertions to be checked dynamically (or statically, if enough information is present). When a program is completed and all uses of **?** have been removed, those same types establish properties that have been statically verified.

Witness Composition: The key to finding the error above was tracking information with witnesses, and combining those witnesses using the composition operator. While that composition was a simple equality check, in general the composed values may be imprecise, and the result is some value that is as precise as both inputs. The information from the witness is used when

[2]Each **?** is actually $?_{@0}$, i.e., annotated with its type's universe level. Our exposition omits levels; we explain them in §3.3.

eliminating an equality proof: when using a witness $\mathbf{t_w}$ of $\mathbf{t_1} =_\mathbf{T} \mathbf{t_2}$ to rewrite a term of type $\mathbf{P(t_1)}$ into $\mathbf{P(t_2)}$, we first cast to $\mathbf{P(t_w)}$, then to $\mathbf{P(t_2)}$. For a program with imprecise types or values, the witness retains the information gained by running the program, preventing unsafe execution, and informing the programmer when a counter-example to an imprecise equality is found.

Here we present an example of a bug that is found, not because of safety, but because a remembered constraint was violated. Consider the following functions:

$$zip : (n : \mathbb{N}) \to FList\ A\ n \to FList\ A\ n \to FList\ (A \times A)\ n$$
$$take : (n : \mathbb{N}) \to (m : \mathbb{N}) \to FList\ B\ (n + m) \to FList\ B\ n$$

Here, *zip* takes two lists of exactly the same length, and produces a list of pairs of their elements, while *take* takes a list with at least n elements, and returns the first n elements of that list. Each function constrains the size of its input, so by tracking equality witnesses, we can also track these constraints and detect where they are incompatible. Now consider lists with imprecise types:

$list_1 := ([1.1, 2.2], refl_?) : FList\ Float\ ?$ $\qquad$ $list_2 := (Cons\ 1.1\ ?, refl_?) : FList\ Float\ ?$

For $list_1$, we are converting a list of length 2 to a fixed-length list of unknown length, since 2 is consistent with ?. For $list_2$, however, the length is truly imprecise, since its tail is the unknown term. We can zip these lists together as $zip\ ?\ list_1\ list_2 : FList\ Float\ ?$, producing another list of unknown length, since recursively applying *zip* to the unknown tail ? produces an unknown result. Applying *take* 3 to the result of *zip* is well typed, since the length ? is consistent with $3 + ?$, i.e., $take\ 3\ ?\ (zip\ ?\ list_1\ list_2) : FList\ Float\ 3$. However, computing the witness flags an error.

This error represents something deeper than a simple safety check: it detects fundamental inconsistencies in statically-determined propositional equalities. In the absence of equality proofs, the call could run safely: $Cons\ (1.1, 3.3)\ ?$ would be a sensible result, having length consistent with 3. The witness composition is not just checking if a list is empty before taking the head, or counting the elements in the list before running *take*. Rather, the information added by *zip*, that the list should have length 2, has been propagated using the $list_1$ witness and composed with the conflicting information. GEq uses witnesses to enforce imprecise equality constraints at run time.

To understand how GEq detects this mismatch, we look at the result of elaborating to CastEq. Initially, $\mathbf{list_2}$ has $1 + ?_\mathbb{N}$ as the witness that $?_\mathbb{N}$ is equal to $1 + ?_\mathbb{N}$. The result of **zip** has 2 as the witness of equality between $?_\mathbb{N}$ and $?_\mathbb{N}$, since that is the length of $\mathbf{list_1}$. This new witness was determined by composition: since $1 + ?_\mathbb{N}$ is consistent with 2, this composition succeeds. (Using a Peano representation of naturals, $\mathbf{S(?)}$ is consistent with $\mathbf{S(S(0))}$) Then, even though **zip**'s result has a type that is consistent with what **take** expects, the run-time type information remembers that **zip** constrained the list to have length 2. The result of **zip** is cast to $\mathbf{FList\ Float\ (3 + ?_\mathbb{N})}$, the type expected by **take** 3. The **zip** result has an equality proof of type $1 + ?_\mathbb{N} =_\mathbb{N} ?_\mathbb{N}$, which is cast to type $1 + ?_\mathbb{N} =_\mathbb{N} 3 + ?_\mathbb{N}$. During this cast, the target value $3 + ?_\mathbb{N}$ is composed with the witness 2. Despite the imprecision, these values are not consistent, and composition produces an error: no value can replace ? to make $\mathbf{S(S(S(?)))}$ equal to $\mathbf{S(S(0))}$. We detail the semantics enabling this in §4.

With equality witnesses, we achieve more than type safety. From the gradual guarantees, we know the above code cannot possibly be made static by replacing the ? uses with static terms. When a witness reduces to an error, the program is equating two terms that are fundamentally not-equal. So the gradual guarantees now inform about equality constraints, in addition to type constraints. These constraints are expressed through types, rather than some external language of assertions.

2.4 Lazily Enforcing Specifications: Function Equalities and Extensionality

Propositional equality is not restricted to first-order values like numbers or to types with decidable equality. In particular, we can form equalities between functions, for which equality is not in general

decidable. The following summarizes how GEq handles propositional equality for functions without encountering the impossibility result of Lennon-Bertrand et al. [2022]. Consider the example they use to show the incompatibility between gradual typing and **refl**-based equality:

$$id_{\mathbb{N}} := (\lambda x.\, x) : \mathbb{N} \to \mathbb{N} \qquad add0 := (\lambda x.\, x + 0) : \mathbb{N} \to \mathbb{N}$$
$$test := \lambda f.\, \mathbf{J}\ (\_.\mathbb{B})\ id_{\mathbb{N}}\ true\ f\ (refl_{id_{\mathbb{N}}}\ :: ?\ ::\ (id_{\mathbb{N}} =_{\mathbb{N}\to\mathbb{N}} f)) : (\mathbb{N} \to \mathbb{N}) \to \mathbb{B}$$

Here **J** is the eliminator for equality: we explain it fully in §4.2, but it suffices to know that in this case, it uses a proof of type $id_{\mathbb{N}} =_{\mathbb{N}\to\mathbb{N}} f$ to rewrite $(\lambda\_.\, \mathbb{B})\ id_{\mathbb{N}}$ to $(\lambda\_.\, \mathbb{B})\ f$. The form $::$ denotes surface type ascriptions, which are elaborated to casts in the cast calculus. Both types reduce to $\mathbb{B}$, but **J** only reduces if the equality proof reduces without error.

Since $id_{\mathbb{N}}$ and $add0$ agree on all inputs, they should be observationally equivalent, producing the same result in any context in which we use them. Violating this would mean that the embedding of CIC into GCIC or GEq does not respect function extensionality, i.e., some statically-equivalent terms are distinguishable in the gradual language. Lennon-Bertrand et al. [2022] offer *test* as a context that distinguishes $id_{\mathbb{N}}$ and $add0$. When $id_{\mathbb{N}}$ is given as an argument, casting $refl_{id_{\mathbb{N}}}$ to $?$ then back to $id_{\mathbb{N}} =_{\mathbb{N}\to\mathbb{N}} id_{\mathbb{N}}$ should produce $refl_{id_{\mathbb{N}}}$. However, when $add0$ is given for f, the cast must fail, since *refl* cannot have type $id_{\mathbb{N}} =_{\mathbb{N}\to\mathbb{N}} add0$.

The first key to avoiding this inequivalence is the witness-based representation of equality. In CastEq, **refl** is the only constructor for equality, but it takes an argument: the consistency witness for the equated terms. Moreover, it does not require the equated terms to be syntactically identical, only that the witness be at least as precise as both of them. So **refl** can have type $\mathbf{id}_{\mathbb{N}} =_{\mathbb{N}\to\mathbb{N}} \mathbf{add0}$.

What witness should be attached to the proof of $\mathbf{id}_{\mathbb{N}} =_{\mathbb{N}\to\mathbb{N}} \mathbf{add0}$? The second key feature is the composition operator of CastEq, which builds said equality witness. Elaborating $refl_{id_{\mathbb{N}}}$ creates witness $\mathbf{id}_{\mathbb{N}}$. The cast to $\mathbf{id}_{\mathbb{N}} =_{\mathbb{N}\to\mathbb{N}} \mathbf{f}$ composes that witness with the destination endpoints, $\mathbf{id}_{\mathbb{N}}$ and $\mathbf{f}$, yielding $\mathbf{id}_{\mathbb{N}}\ \&_{\mathbb{N}\to\mathbb{N}}\ \mathbf{id}_{\mathbb{N}}\ \&_{\mathbb{N}\to\mathbb{N}}\ \mathbf{f}$. The semantics of $\&$ reduce this composition to $\lambda \mathbf{x}.\ (\mathbf{x}\ \&_{\mathbb{N}}\ \mathbf{x}\ \&_{\mathbb{N}}\ (\mathbf{f}\ \mathbf{x}))$, similar to how a higher-order contract applied to a function produces a new function that checks the input and result [Findler and Felleisen 2002]. Since $\&$ is an operator in the language, the composition does not need to reduce further, but when the function is applied, it continues to reduce. The same holds when we replace $\mathbf{f}$ with $\mathbf{id}_{\mathbb{N}}$ or $\mathbf{add0}$. Section 6 defines precision such that $\mathbf{x}\ \&_{\mathbb{N}}\ \mathbf{x}\ \&_{\mathbb{N}}\ \mathbf{f}\ \mathbf{x}$ is more precise than both $\mathbf{x}$ and $\mathbf{f}\ \mathbf{x}$, so the above composition is a valid witness. We define semantics for **J** so that when it is given the equality proof with the above witness, it reduces, so **test** reduces to **true** for both $\mathbf{id}_{\mathbb{N}}$ and $\mathbf{add0}$.

How can equating these functions be safe, since deferring composition means that we can prove an equality between unequal functions? As we saw with **sort** above, **J** casts through the witness, so when functions are extensionally non-equal, trying to prove equality between their results dynamically fails. Consider instead $\mathbf{subadd1} := (\lambda(\mathbf{x} : \mathbb{N}).\, \mathbf{x} - 1 + 1)$. Since $0 - \mathbf{t} = 0$ in $\mathbb{N}$, $\mathbf{subadd1}\ 0 = 1$. We can use the witness $\lambda \mathbf{x}.\ (\mathbf{x}\ \&_{\mathbb{N}}\ \mathbf{x}\ \&_{\mathbb{N}}\ (\mathbf{subadd1}\ \mathbf{x}))$ to inhabit $\mathbf{id}_{\mathbb{N}} =_{\mathbb{N}\to\mathbb{N}} \mathbf{subadd1}$. However, if we try to use **J** to prove that $\mathbf{id}_{\mathbb{N}}\ 0 =_{\mathbb{N}} \mathbf{subadd1}\ 0$, the result substitutes 0 for $\mathbf{x}$ in the witness, giving $0\ \&_{\mathbb{N}}\ 0\ \&_{\mathbb{N}}\ 1$, which reduces to an error.

The consequence of our approach is that GEq supports a limited form of extensionality. *Neutral terms*, i.e., variables or terms for which reduction is blocked by one or more variables, always compose to a non-error, so we can build a witness capturing the plausibility of equality between them, given partial information. That witness makes an equality proof constructible. Furthermore, any two functions with neutral bodies compose to a non-error. If the functions agree on all inputs, eliminating their equality never fails and the proof of equality can be freely used. If the functions disagree on some input, an error is raised when building a term that relies on the functions producing the same value for said input. Since it is undecidable whether two functions agree on every input, this approach finds a balance between decidability and flexibility.

Static vs. Dynamic Consistency: For non-dependent gradual types, the successful composition of two types usually implies that they are consistent. However, for GEq, two neutrals always compose to a more-precise term. To conservatively extend CIC, all ill-typed (fully-static) CIC programs must be ill-typed in GEq, and an uninhabited CIC type must not be inhabited by any fully-static GEq terms. So GEq cannot have all neutrals consistent, since this would yield a fully-static proof of $(\lambda \mathbf{x} \lambda \mathbf{y}.\, \mathbf{x}) =_{\mathbf{T}\to\mathbf{T}\to\mathbf{T}} (\lambda \mathbf{x} \lambda \mathbf{y}.\, \mathbf{y})$.

We resolve this tension with separate static and dynamic notions of consistency (§6). Terms are statically consistent if they are syntactically equal up to α-equivalence, reduction, and occurrences of $?_{\mathbf{T}}$. Terms are dynamically consistent if they compose without error, or equivalently, if there exists a non-error term as precise as both terms. Essentially, terms are dynamically-consistent if they are statically consistent in the non-neutral parts. The type rules for GEq use static consistency. Some pairs of terms are not statically consistent, yet still compose to a non-error term.

To compare static and dynamic consistency, consider the ill-typed CIC term $\mathsf{refl} : \mathsf{x} =_{\mathbb{N}} \mathsf{y}$. When embedded into GEq, $\mathit{refl} : x =_{\mathbb{N}} y$ is still ill-typed: the expected type of $x =_{\mathbb{N}} y$ and the actual type of $x =_{\mathbb{N}} x$ are not statically consistent, because the variables x and y are not identical. In CastEq, $\mathbf{x}$ and $\mathbf{y}$ are neutral, and hence dynamically consistent, meaning $\mathbf{x} \mathbin{\&_{\mathbb{N}}} \mathbf{y}$ witnesses $\mathbf{x} =_{\mathbb{N}} \mathbf{y}$.

Allowing neutrals to be dynamically consistent does not interfere with conservatively extending CIC. For conservative extension, every ill-typed CIC program should be ill-typed in GEq. In the absence of $?$, pairs of definitionally-unequal CIC terms are statically inconsistent. While GEq gives refl the same type as CIC, CastEq lets $\mathbf{refl}$ prove equality for dynamically consistent terms. However, dynamic consistency does not allow CastEq to type ill-typed CIC terms, because CIC programs are elaborated into a subset of CastEq where $\mathbf{t}$ only witnesses $\mathbf{t} =_{\mathbf{T}} \mathbf{t}$ and all casts have the form $\langle \mathbf{T} \Leftarrow \mathbf{T} \rangle$. The type $(\mathsf{x} : \mathbb{N}) \to (\mathsf{y} : \mathbb{N}) \to (\mathsf{x} =_{\mathbb{N}} \mathsf{y})$ is uninhabited in CIC, and while this type is inhabited in CastEq using witness $\mathbf{x} \mathbin{\&_{\mathbb{N}}} \mathbf{y}$, the use of $\&_{\mathbb{N}}$ puts the witness outside the static fragment of CastEq. The term $\mathbf{x} \mathbin{\&_{\mathbb{N}}} \mathbf{y}$ does not correspond to any typed or ill-typed CIC program.

Static and dynamic consistency let us balance conflicting goals. If all statically inconsistent functions composed to an error, then statically-equivalent terms would not be gradually equivalent, making it harder to reason about program equivalence. Using dynamic consistency during type checking would not conservatively extend CIC. By separating these, we obtain conservative extension, while dynamically respecting all extensional equalities.

3 THE STATIC LANGUAGE AND GCIC

To begin our development, we review the state-of-the-art for handling inductive types in a gradual language. We describe the Bidirectional CIC (BCIC), a modification of CIC whose bidirectional types are convenient for gradual typing [Lennon-Bertrand 2021]. We then describe the gradual surface language GCIC, along with the cast calculus, CastCIC, and a translation from GCIC to CastCIC [Lennon-Bertrand et al. 2022]. Specifically, we use $\mathrm{GCIC}^{\mathcal{G}}$, the variant that satisfies the gradual guarantees and embeds CIC, but sacrifices strong normalization. We discuss options for decidable type checking in §7.2.1. Though GCIC is not a contribution of this paper, we use it as the starting point for our development, making additions to the surface language and cast calculus.

3.1 Bidirectional CIC

3.1.1 Syntax. Figure 2 gives the *bidirectional calculus of inductive constructions* (BCIC) as originally presented by Lennon-Bertrand [2021], though we modify their notation to maximize clarity for GEq's additions. BCIC terms are denoted by metavariables t and T, loosely following the convention that T be reserved for types. Variables are denoted by $\mathsf{x}, \mathsf{y}, \mathsf{z}$. BCIC has variables, a predicative hierarchy of universes, function types, functions, and applications. Technically, BCIC extends the predicative, non-cumulative fragment of CIC: each function type is in a higher universe than its

$$t, T ::= x \mid \mathbf{Type}_i \mid (x : T_1) \to T_2 \mid \lambda(x : T).\, t \mid t_1\ t_2 \mid C@\{i\}(\overline{t}) \mid D^C@\{i\}(\overline{t}, \overline{t'}) \mid \mathsf{ind}_C(t_1, z.T, x.\overline{y.t_2})$$

$$h ::= C \mid \mathbf{Type} \mid \Pi$$

$\Gamma \vdash t \Rightarrow T$ (Synthesis) $\quad \Gamma \vdash t \Leftarrow T$ (Checking) $\quad \Gamma \vdash t \Rightarrow_h T$ (Constrained Synthesis)

$$\frac{\Gamma \vdash t \Rightarrow T' \qquad T \longrightarrow^* T'' \qquad T' \longrightarrow^* T''}{\Gamma \vdash t \Leftarrow T} \qquad \frac{(x : T) \in \Gamma}{\Gamma \vdash x \Rightarrow T} \qquad \frac{}{\Gamma \vdash \mathbf{Type}_i \Rightarrow \mathbf{Type}_{i+1}}$$

$$\frac{\Gamma \vdash T_1 \Rightarrow_{\mathbf{Type}} \mathbf{Type}_i \quad \Gamma, x : T_1 \vdash T_2 \Rightarrow_{\mathbf{Type}} \mathbf{Type}_j}{\Gamma \vdash (x : T_1) \to T_2 \Rightarrow \mathbf{Type}_{\max(i,j)}} \qquad \frac{\Gamma \vdash t_0 \Rightarrow_\Pi (x : T_1) \to T_2 \quad \Gamma \vdash t_1 \Leftarrow T_1}{\Gamma \vdash t_0\ t_1 \Rightarrow [t_1/x]T_2} \qquad \frac{\Gamma \vdash T_1 \Rightarrow_{\mathbf{Type}} \mathbf{Type}_i \quad \Gamma, x : T_1 \vdash t \Rightarrow T_2}{\Gamma \vdash \lambda(x : T_1).\, t \Rightarrow (x : T_1) \to T_2}$$

$$\frac{\Gamma \vdash t_k \Leftarrow \mathbf{Params}_k(C, i)[\overline{t}]}{\Gamma \vdash C@\{i\}(\overline{t}) \Rightarrow \mathbf{Type}_i} \qquad \frac{\Gamma \vdash t_k \Leftarrow \mathbf{Params}_k(C, i)[\overline{t}] \qquad \Gamma \vdash t'_m \Leftarrow \mathbf{Args}_m(C, i, D)[\overline{t}, \overline{t'}]}{\Gamma \vdash D^C@\{i\}(\overline{t}, \overline{t'}) \Rightarrow C@\{i\}(\overline{t})}$$

$$\frac{\Gamma \vdash t_{scrut} \Rightarrow_C C@\{i\}(\overline{t_{par}}) \quad \Gamma, z : C(\overline{t_{par}}) \vdash T_P \Rightarrow_{\mathbf{Type}} \mathbf{Type}_j \quad \left(\Gamma, x_{rec} : X, \overline{y} : \mathbf{Args}(C, i, D_k)[\overline{t_{par}}, \overline{y}] \vdash t_{rhs_k} \Leftarrow [D_k^C@\{i\}(\overline{t_{par}}, \overline{y})/z]T_P\right)^k \quad \textit{where } X := (z : C@\{i\}(\overline{t_{par}})) \to T_P}{\Gamma \vdash \mathsf{ind}_C(t_{scrut}, z.T_P, x_{rec}.\overline{y}.\overline{t_{rhs}}) \Rightarrow [t_{scrut}/z]T_P} \qquad \frac{\Gamma \vdash t \Rightarrow T \quad T \longrightarrow^* \mathbf{Type}_i}{\Gamma \vdash t \Rightarrow_{\mathbf{Type}} \mathbf{Type}_i}$$

$$\frac{\Gamma \vdash t \Rightarrow T \qquad T \longrightarrow^* (x : T_1) \to T_2}{\Gamma \vdash t \Rightarrow_\Pi (x : T_1) \to T_2} \qquad \frac{\Gamma \vdash t \Rightarrow T \qquad T \longrightarrow^* C@\{i\}(\overline{t_p})}{\Gamma \vdash t \Rightarrow_C C@\{i\}(\overline{t_p})}$$

$t \rightsquigarrow t'$ (Notions of Reduction) $\quad t \longrightarrow t'$ (Contextual reduction)

$$(\lambda(x : T).\, t)\ t' \rightsquigarrow [t'/x]t \qquad \frac{t \rightsquigarrow t' \qquad C \text{ an arbitrary context}}{C[t] \longrightarrow C[t']}$$

$$\mathsf{ind}_C(D_k(\overline{t_{par}}, \overline{t_{arg}}), z.T_P, x_{rec}.\overline{y.t_{rhs}}) \rightsquigarrow [\lambda(x : C(\overline{t_{par}})).\, \mathsf{ind}_C(x, z.T, x_{rec}.\overline{y.t_{rhs}})/x_{rec}][\overline{t_{arg}}/\overline{y}]t_{rhs_k}$$

Fig. 2. Bidirectional CIC: Syntax, Typing and Semantics

domain and codomain, and there is no subtyping between universe levels. We assume a pre-existing set of inductive type constructors, denoted by the metavariable C, each of which has a fixed set of data constructors D^C. Type and data constructors are annotated with the level of their type, though we omit these annotations when they are not relevant.

To eliminate members of inductive types, a combined form $\mathsf{ind}_C(t_1, z.T, x.\overline{y.t_2})$ replaces CIC's `fix` and `match`. This form branches on the scrutinee t_1 and has a parameterized result type T, called the *motive* [McBride 2002], that binds a variable of the scrutinee's type. The branches $\overline{t_2}$ correspond to the constructors $\overline{D^C}$ of C. In each branch, the variables $\overline{y}$ are bound to the arguments to D^C, and x refers to the whole ind_C expression, to facilitate recursion. The ind_C form expresses an induction principle: if each branch produces a result of type T where z is bound to D^C applied to $\overline{y}$, the elimination has type T where z is bound to the scrutinee t_1. In essence, ind_C says that if we can build a T for each constructor D^C of $C(\overline{t_{par}})$, then we can build one for any value of $C(\overline{t_{par}})$. Normally, a separate check ensures that recursive calls are only made on structurally smaller

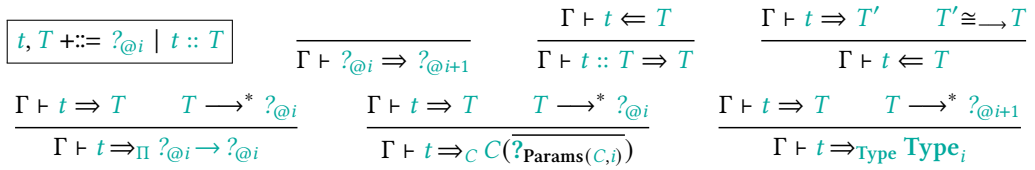

Fig. 3. GCIC: Syntax and Typing Lemmas

arguments, but we omit this check, since it is orthogonal to gradual typing and GEq would not be strongly normalizing even with it.

BCIC also uses *head tags*, denoted by h, which act as symbols to specify a type constructor without specifying its arguments. We use these in typing, e.g. for expressing that an applied function must synthesize a function type, even though we do not know what the domain and codomain should be. Tags are also useful in GCIC for defining the least precise type with a given head.

3.1.2 Typing and Semantics. The typing and semantics (Fig. 2) for BCIC resemble the typical presentation of CIC, but typing is divided into synthesis, which produces a type, and checking, which consumes a type. The semantics is given with primitive notions of reduction $\rightsquigarrow$, contextual stepping $\longrightarrow$ where any sub-term reduces (even under binders), and multi-step contextual reduction $\longrightarrow^*$, which allows zero or more steps with $\longrightarrow$. Because function types bind their parameter in the codomain, applications synthesize a type depending on the value of the argument, since it is substituted for x in the codomain type. A term checks against any type that reduces to the same type as its synthesized type, since an application may have produced a type that must be reduced before comparing. Constrained-synthesis, $\Gamma \vdash t \Rightarrow_h T$, generalizes the pattern of synthesizing a type for a term after reducing it to a point that it has the desired head h. Function application has the standard β-reduction rule.

For inductive types, the typing rule establishes that, if T_P is a type parameterized over a value x from the inductive type C, and we can (recursively) build a T_P for each constructor of C, then we can build a T_P for any member or C. Hence $\mathsf{ind}_C()$ form gives an induction principle for C, hence the notation $\mathsf{ind}_C(\ldots)$. Inductive types may be parameterized, but each constructor has the same return type. The reduction rule says that an $\mathsf{ind}_C(\ldots)$ form given a value $D^C(\ldots)$ reduces to the branch corresponding to D^C.

3.2 GCIC: The Surface Language

Figure 3 extends BCIC into GCIC, the Gradual CIC, by adding the *imprecise term* $?_{@i}$, which can be used at any type in universe level i, along with type ascriptions, which were not in BCIC because all forms synthesized types. We use $?_T$ as sugar for $?_{@i}$ when $T : \mathbf{Type}_i$.

Dependent types complicate the typing of GCIC. Because the dynamic semantics of GCIC are defined using a cast calculus, and typing refers to reduction of terms, Lennon-Bertrand et al. [2022] define GCIC typing with cast calculus types. Nevertheless, we can establish lemmas (Fig. 3), phrased like rules, which provide intuition for how GCIC terms are typed against GCIC types, helping GCIC be understood without diving into the details of the cast calculus.

The unknown term $?_{@i}$ synthesizes $?_{@i+1}$ i.e. its type is unknown, one level up in the universe hierarchy. A term checks against any type consistent with its synthesized type, where the relation $\cong_{\rightarrow}$ is understood to mean convertibility up to well-typed occurrences of ? (explained fully in §6.1.2). An ascribed term synthesizes the given type if it checks against it, relaxing or tightening the types of gradual terms. Constrained synthesis accounts for terms synthesizing $?_{\mathbf{Type}_i}$ by producing the *germ* (called the *ground type* in non-dependently typed literature). The type $\mathrm{germ}_i(h)$ is the

$$\boxed{\mathbf{t}, \mathbf{T} ::= ?_{\mathbf{T}} \mid \mathcal{U}_{\mathbf{T}} \mid \langle \mathbf{T}_2 \Leftarrow \mathbf{T}_1 \rangle \mathbf{t}}$$

$$\frac{\Gamma \vdash \mathbf{T} \Rightarrow_{\mathbf{Type}} \mathbf{Type}_i}{\Gamma \vdash ?_{\mathbf{T}} \Rightarrow \mathbf{T}} \qquad \frac{\Gamma \vdash \mathbf{T} \Rightarrow_{\mathbf{Type}} \mathbf{Type}_i}{\Gamma \vdash \mathcal{U}_{\mathbf{T}} \Rightarrow \mathbf{T}} \qquad \frac{\Gamma \vdash \mathbf{T}_j \Rightarrow_{\mathbf{Type}} \mathbf{Type}_i \text{ for } j \in \{1,2\} \qquad \Gamma \vdash \mathbf{t} \Leftarrow \mathbf{T}_1}{\Gamma \vdash \langle \mathbf{T}_2 \Leftarrow \mathbf{T}_1 \rangle \mathbf{t} \Rightarrow \mathbf{T}_2}$$

Propogation Reductions:

RedPropCastInd(Unk,Err)

$$\langle \mathbf{C}(\overline{\mathbf{t}_2}) \Leftarrow \mathbf{C}(\overline{\mathbf{t}_1}) \rangle ?_{\mathbf{T}_1} \rightsquigarrow ?_{\mathbf{C}(\overline{\mathbf{t}_2})}$$
$$\langle \mathbf{C}(\overline{\mathbf{t}_2}) \Leftarrow \mathbf{C}(\overline{\mathbf{t}_1}) \rangle \mathcal{U}_{\mathbf{T}_1} \rightsquigarrow \mathcal{U}_{\mathbf{C}(\overline{\mathbf{t}_2})}$$

RedPropCastDown(Unk,Err)

$$\langle \mathbf{T} \Leftarrow ?_{\mathbf{Type}_i} \rangle ?_{?_{\mathbf{Type}_i}} \rightsquigarrow ?_{\mathbf{T}}$$
$$\langle \mathbf{T} \Leftarrow ?_{\mathbf{Type}_i} \rangle \mathcal{U}_{?_{\mathbf{Type}_i}} \rightsquigarrow \mathcal{U}_{\mathbf{T}}$$

RedPropFun(Unk,Err)

$$?_{(\mathbf{x}:\mathbf{T}_1)\to\mathbf{T}_2} \rightsquigarrow \lambda(\mathbf{x} : \mathbf{T}_1).\, ?_{\mathbf{T}_2}$$
$$\mathcal{U}_{(\mathbf{x}:\mathbf{T}_1)\to\mathbf{T}_2} \rightsquigarrow \lambda(\mathbf{x} : \mathbf{T}_1).\, \mathcal{U}_{\mathbf{T}_2}$$

RedPropMatch(Unk,Err)

$$\mathbf{ind}_{\mathbf{C}}(?_{\mathbf{C}(\overline{\mathbf{t}_{\mathbf{par}}})}, \mathbf{z}.\mathbf{T}_{\mathbf{P}}, \mathbf{x}_{\mathbf{f}}.\overline{\mathbf{y}}.\overline{\mathbf{t}}) \rightsquigarrow ?_{[?_{\mathbf{C}(\overline{\mathbf{t}_{\mathbf{par}}})}/\mathbf{z}]\mathbf{T}_{\mathbf{P}}}$$
$$\mathbf{ind}_{\mathbf{C}}(\mathcal{U}_{\mathbf{C}(\overline{\mathbf{t}_{\mathbf{par}}})}, \mathbf{z}.\mathbf{T}_{\mathbf{P}}, \mathbf{x}_{\mathbf{f}}.\overline{\mathbf{y}}.\overline{\mathbf{t}}) \rightsquigarrow \mathcal{U}_{[\mathcal{U}_{\mathbf{C}(\overline{\mathbf{t}_{\mathbf{par}}})}/\mathbf{z}]\mathbf{T}_{\mathbf{P}}}$$

Cast Reductions:

RedCastUpDown

$$\langle \mathbf{T} \Leftarrow ?_{\mathbf{Type}_i} \rangle \langle ?_{\mathbf{Type}_i} \Leftarrow \mathrm{germ}_i(\mathbf{h}) \rangle \mathbf{t} \rightsquigarrow \langle \mathbf{T} \Leftarrow \mathrm{germ}_i(\mathbf{h}) \rangle \mathbf{t}$$

RedCastInd

$$\langle \mathbf{C}(\overline{\mathbf{t}'_{\mathbf{par}}}) \Leftarrow \mathbf{C}(\overline{\mathbf{t}_{\mathbf{par}}}) \rangle \mathbf{D}^{\mathbf{C}}(\overline{\mathbf{t}_{\mathbf{par}''}}, \overline{\mathbf{t}_{\mathbf{arg}}}) \rightsquigarrow \mathbf{D}^{\mathbf{C}}(\overline{\mathbf{t}'_{\mathbf{par}}}, \overline{\mathbf{t}'_{\mathbf{arg}}})$$
$$\textit{where } \mathbf{t}'_{\mathbf{arg}_i} := \langle \mathbf{Args}_i(\mathbf{C}, \mathbf{i}, \mathbf{D})[\overline{\mathbf{t}'_{\mathbf{par}}}, \overline{\mathbf{t}'_{\mathbf{args}}}] \Leftarrow \mathbf{Args}_i(\mathbf{C}, \mathbf{i}, \mathbf{D})[\overline{\mathbf{t}_{\mathbf{par}}}, \overline{\mathbf{t}_{\mathbf{args}}}] \rangle \mathbf{t}_{\mathbf{arg}_i}$$

RedCastType

$$\langle \mathbf{Type}_i \Leftarrow \mathbf{Type}_i \rangle \mathbf{T} \rightsquigarrow \mathbf{T}$$

RedCastDomErr

$$\langle \mathbf{T} \Leftarrow \mathcal{U}_{\mathbf{Type}_i} \rangle \mathbf{t} \rightsquigarrow \mathcal{U}_{\mathbf{T}}$$

RedCastCodomErr

$$\langle \mathcal{U}_{\mathbf{Type}_i} \Leftarrow \mathbf{T} \rangle \mathbf{t} \rightsquigarrow \mathcal{U}_{\mathcal{U}_{\mathbf{Type}_i}}$$

RedCastHeadErr

$$\frac{\mathrm{head}(\mathbf{T}) \neq \mathrm{head}(\mathbf{T}')}{\langle \mathbf{T}' \Leftarrow \mathbf{T} \rangle \mathbf{t} \rightsquigarrow \mathcal{U}_{\mathbf{T}'}}$$

RedCastFun

$$\langle (\mathbf{x} : \mathbf{T}'_1) \to \mathbf{T}'_2 \Leftarrow (\mathbf{x} : \mathbf{T}_1) \to \mathbf{T}_2 \rangle \mathbf{t} \rightsquigarrow \lambda \mathbf{y}.\, \langle \mathbf{T}'_2 \Leftarrow [\langle \mathbf{T}_1 \Leftarrow \mathbf{T}'_1 \rangle \mathbf{y}/\mathbf{x}]\mathbf{T}_2 \rangle (\mathbf{t}\, \langle \mathbf{T}_1 \Leftarrow \mathbf{T}'_1 \rangle \mathbf{y})$$

RedCastFunGerm

$$\frac{(\mathbf{x} : \mathbf{T}_1) \to \mathbf{T}_2 \neq \mathrm{germ}_j(\Pi) \text{ for } j \geq i}{\langle ?_{\mathbf{Type}_i} \Leftarrow (\mathbf{x} : \mathbf{T}_1) \to \mathbf{T}_2 \rangle \mathbf{t} \rightsquigarrow \langle ?_{\mathbf{Type}_i} \Leftarrow ?_{\mathbf{Type}_i} \to ?_{\mathbf{Type}_i} \rangle \langle ?_{\mathbf{Type}_i} \to ?_{\mathbf{Type}_i} \Leftarrow (\mathbf{x} : \mathbf{T}_1) \to \mathbf{T}_2 \rangle \mathbf{t}}$$

RedCastIndGerm

$$\frac{\mathbf{C}(\overline{\mathbf{t}_{\mathbf{par}}}) \neq \mathrm{germ}_j(\mathbf{C}) \text{ for } j \geq i}{\langle ?_{\mathbf{Type}_i} \Leftarrow \mathbf{C}(\overline{\mathbf{t}_{\mathbf{par}}}) \rangle \mathbf{t} \rightsquigarrow \langle ?_{\mathbf{Type}_i} \Leftarrow \mathbf{C}(\overline{?_{\mathbf{Params}(\mathbf{C})}}) \rangle \langle \mathbf{C}(\overline{?_{\mathbf{Params}(\mathbf{C})}}) \Leftarrow \mathbf{C}(\overline{\mathbf{t}_{\mathbf{par}}}) \rangle \mathbf{t}}$$

Fig. 4. CastCIC: Typing and Reduction rules that extend BCIC

least precise type with a given head in universe i. For function types, the germ is $?_{@i} \to ?_{@i}$, and $\mathbf{Type}_i$ is its own germ. For inductives, the germ is $C(\overline{?_{@i}})$ where the i's are the parameters' levels.

3.3 CastCIC: The Cast Calculus

3.3.1 Syntax, Typing and Reductions. Figure 4 presents CastCIC, the cast calculus for GCIC. CastCIC extends BCIC with the unknown term $?$, an error $\mathcal{U}$, and a cast $\langle \mathbf{T}_2 \Leftarrow \mathbf{T}_1 \rangle \mathbf{t}$ from type $\mathbf{T}_1$ to $\mathbf{T}_2$. Forms $?_{\mathbf{T}}$ and $\mathcal{U}_{\mathbf{T}}$ are ascribed with their type $\mathbf{T}$, which affect the dynamic semantics of CastCIC. The CastCIC type system contains all the rules from Fig. 2 plus the rules of Fig. 4. Terms $?_T$ and $\mathcal{U}_T$ synthesize their ascribed type $\mathbf{T}$, while casts synthesize the destination type, given that the term being cast checks against the source type, and that both types are well-formed. Because casts are explicit, the checking rule uses definitional equality, rather than consistency.

The CastCIC semantics includes all BCIC reductions, plus cast rules and "propagation rules" that handle $?$ and $\mathcal{U}$. At type $(\mathbf{x} : \mathbf{T}_1) \to \mathbf{T}_2$, $?$ and $\mathcal{U}$ expand to $\lambda(\mathbf{x} : \mathbf{T}_1).\, ?_{\mathbf{T}_2}$ in RedPropFunUnk and $\lambda(\mathbf{x} : \mathbf{T}_1).\, \mathcal{U}_{\mathbf{T}_2}$ in RedPropFunErr. In the remaining RedProp rules, eliminating or casting $?$ or $\mathcal{U}$ produces $?$ or $\mathcal{U}$. Cast rules either convert between types with the same head, cast to

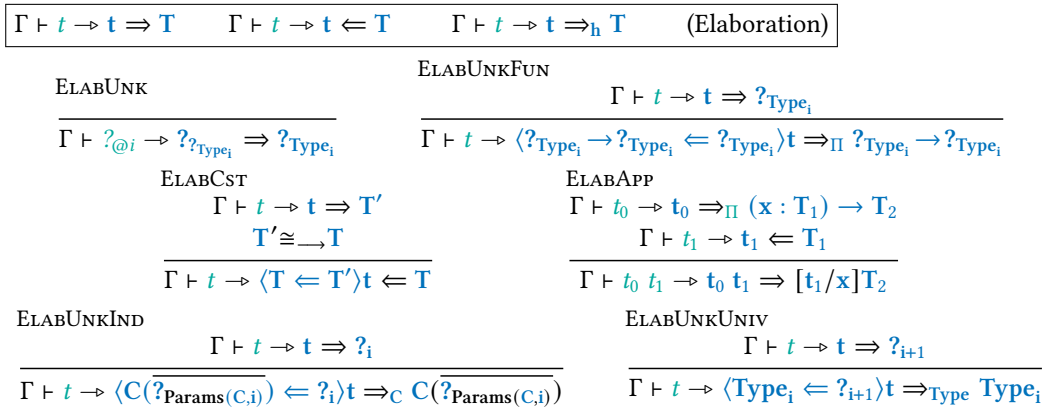

Fig. 5. Elaboration from GCIC to CastCIC (homomorphic rules omitted)

$?_{\mathbf{Type}_\ell}$, or produce an error. A cast from $\mathbf{Type}_i$ to itself reduces away (RedCastType). For inductives, casts from $\mathbf{C}(\overline{\mathbf{t}_1})$ to $\mathbf{C}(\overline{\mathbf{t}_2})$ reduce by casting arguments to their new types (RedCastInd). Note that $\mathbf{t}_{par}$ and $\mathbf{t}_{par''}$ need not match, but typing guarantees that they are convertible. Casts between types with mismatched heads produce an error (RedCastHeadErr), as do casts to or from $\mho_{\mathbf{Type}_i}$ (RedCastDomErr, RedCastCodomErr). A cast from the germ for a given head does *not* reduce: $\langle ?_{\mathbf{Type}_i} \Leftarrow \mathrm{germ}(\mathbf{T}) \rangle$ acts as a tag, injecting into $?_{\mathbf{Type}_i}$. Casts from non-germ types to $?_{\mathbf{Type}_i}$ decompose into casts through the germ that are then tagged with their injection into $?_{\mathbf{Type}_i}$ (RedCastFunGerm, RedCastIndGerm). In RedCastUpDown, a cast *from* $?_{\mathbf{Type}_i}$ to $\mathbf{T}$ reduces when the value being cast originates from a type with a matching head, and was accordingly tagged with a cast from $\mathrm{head}(\mathbf{T})$ to $?_{\mathbf{Type}_i}$.

3.3.2 Elaboration. Finally, elaboration (Fig. 5) defines the relationship between GCIC and CastCIC. Like CastCIC, elaboration has synthesis, checking, and constrained synthesis, but each produces the elaboration of the subject term as output. ElabUnk synthesizes the unknown type for the unknown term at the given universe level. ElabApp works like a normal dependent function application, but uses the elaboration of the argument to replace the parameter in the return type. ElabCst checks a term against a type consistent with its synthesized type, inserting the cast between these types into the elaboration. Figure 5 also defines new constrained synthesis rules. Rule ElabUnkFun works like the corresponding lemma, but adds the necessary cast to the elaborated term. Rules ElabUnkInd and ElabUnkUniv work similarly. We omit the elaboration rules corresponding to the remaining BCIC rules, which homomorphically elaborate the sub-terms of a given term.

Elaboration defines GCIC typing: we say $t : T$ when $\cdot \vdash T \rightharpoonup \mathbf{T} \Rightarrow_{\mathbf{Type}} \mathbf{Type}_i$ and $\cdot \vdash t \rightharpoonup \mathbf{t} \Leftarrow \mathbf{T}$.

4 PROPOSITIONAL EQUALITY

The main contribution of our paper is GEq: an extension of GCIC with propositional equality, where the information about an imprecise value accumulates at run time to detect inconsistencies. We define GEq's semantics using a cast calculus CastEq, which extends CastCIC with equality.

The core idea is that a surface-language proof of type $t_1 =_T t_2$ is elaborated into a witness for the consistency of $\mathbf{t}_1$ and $\mathbf{t}_2$. Much like evidence[3] from AGT [Garcia et al. 2016] or the middle-type from threesomes [Siek and Wadler 2010], the consistency witness between terms is a term that is at least as precise as either term. The standard equality proof, $refl_t : t =_T t$, witnesses that t is

[3]Evidence is more complex in AGT, since it can witness subtyping. Evidence for plausible equality between types collapses to a single term as precise as the equated terms.

consistent with itself, while the imprecise proof $?_{t_1 =_T t_2}$ is witnessed by the least precise term that is dynamically consistent with $\mathbf{t}_1$ and $\mathbf{t}_2$. As a program runs, equality witnesses may take values between these extremes, which may be more precise than the witness for **refl** when t is imprecise.

The technical challenge with adding propositional equality is determining how to combine information represented by the equality witnesses. When casting between types $\mathbf{t}_1 =_\mathbf{T} \mathbf{t}_2$ to $\mathbf{t}'_1 =_\mathbf{T} \mathbf{t}'_2$, both of which may be imprecise, we must transform a witness $\mathbf{t}_\mathbf{w}$ for $t_1 =_T t_2$ to one for $t'_1 =_T t'_2$, but even though $\mathbf{t}_\mathbf{w}$ is as precise as $\mathbf{t}_1$ and $\mathbf{t}_2$, it may not be as precise as $\mathbf{t}'_1$ and $\mathbf{t}'_2$. So we need a composition operator that can take $\mathbf{t}'_1$, $\mathbf{t}'_2$ and $\mathbf{t}_\mathbf{w}$ and produce a term that is as precise as all three. However, to respect static observational equivalences and avoid the problems of §2.4, composition cannot be a syntactic meta-operation. The issue is with neutral terms, i.e., variables, or terms whose reduction is blocked by applying or eliminating a variable. Syntactic composition would require distinct neutral terms to compose to an error, but that would violate extensionality.

Along with composition, we must define a notion of precision that determines valid witnesses of consistency. For the evolution of type information to be monotone, the operator $\&$ should compute a lower bound with respect to this notion of precision. Computing the greatest lower bound prevents premature errors, although the proof that composition is the greatest lower bound is left to future work. With non-dependent gradual types, precision can be syntactically, by adding structural rules to $\mathbf{t} \sqsubseteq ?_\mathbf{T}$, but structural rules are not flexible enough to handle composition.

The solutions to these two challenges are interdependent. We avoid the issues with syntactic composition by adding it as a separate syntactic construct to CastEq, so that composition of neutral terms is itself a neutral term. However, if composition is a construct in CastEq, then precision must be defined to accommodate terms that feature composition, so composing two neutral terms produces something that is actually as precise as those two terms. Precision must be defined to respect composition without losing its other important properties, such as transitivity.

This section gives typing and semantics for gradual propositional equality, where proofs of equality are represented by consistency witnesses, but we leave the exact definitions of consistency and precision unspecified. In §5, we describe the properties that consistency and precision should fulfill to ensure that GEq satisfies type safety and the gradual guarantees. Finally, §6 instantiates GEq with notions of precision and consistency that fulfill our goals while ensuring decidable consistency-checking. We separate our presentation in this way to motivate the choices we make in the design of precision, and to avoid monolithic proofs when developing GEq's metatheory.

We write precision as $\Gamma_1 | \Gamma_2 \vdash \mathbf{t}_1 \sqsubseteq^{\longleftarrow}_{\longrightarrow} \mathbf{t}_2$ and consistency as $\mathbf{t}_1 \cong_{\longrightarrow} \mathbf{t}_2$, highlighting the operators in grey to indicate that their definitions are not yet specified. The subscript $\longrightarrow$ on $\cong_{\longrightarrow}$ indicates that it is *definitional consistency*, whose name is chosen by analogy to definitional equality, since the operands can be reduced before being compared structurally [Martin-Löf 1975]. Precision is *precision modulo conversion*, meaning it is closed under the equivalence relation given by convertibility. Unlike consistency, precision modulo conversion can look backwards in time, relating terms that are the *results* of reducing syntactically-related terms, in addition to relating terms that are syntactically-related after reducing. We discuss the need for this in §4.2. Precision takes two contexts, as its operands must be typed in different contexts.

4.1 GEq Syntax and Typing

Fig. 6 extends GCIC to GEq by adding the equality type, introduction form, and eliminator. Their types are identical to what is expected in the static setting. Again, because surface typing is defined by elaboration, the given rules are actually admissible lemmas. An equality type $t_1 =_T t_2$ denotes equality between any two values of consistent types (because each endpoint is checked against T). The reflexive proof $refl_t$ synthesizes type $t =_T t$, so long as t is well typed at type T. The eliminator

$$\boxed{t +::= t =_T t \mid \mathit{refl}_t \mid \mathbf{J}(x.T, t_1, t_2, t_3, t_4)}$$
$$\boxed{h +::= =}$$

$$\frac{\Gamma \vdash t_{eq} \Rightarrow_= t_1 =_T t_2 \qquad \Gamma, x : T \vdash T_P \Rightarrow_{\mathbf{Type}} \mathbf{Type}_i \quad \Gamma \vdash t_1 \Leftarrow T \qquad \Gamma \vdash t_2 \Leftarrow T \qquad \Gamma \vdash t_{P1} \Leftarrow [t_1/x]T_P}{\Gamma \vdash \mathbf{J}(x.T_P, t_1, t_2, t_{P1}, t_{eq}) \Rightarrow [t_2/x]T_P}$$

$$\frac{\Gamma \vdash t_1 \Leftarrow T \qquad \Gamma \vdash t_2 \Leftarrow T \qquad \Gamma \vdash T \Rightarrow_{\mathbf{Type}} \mathbf{Type}_i}{\Gamma \vdash t_1 =_T t_2 \Rightarrow \mathbf{Type}_i} \qquad \frac{\Gamma \vdash t \Rightarrow T}{\Gamma \vdash \mathit{refl}_t \Rightarrow t =_T t}$$

Fig. 6. GEq: Syntax and Typing Lemmas

$$\boxed{\mathbf{t} +::= \mathbf{t}_1 =_\mathbf{T} \mathbf{t}_2 \mid \mathbf{refl(t)}_{\vdash \mathbf{t}_1 \cong \mathbf{t}_2} \mid \mathbf{J(x.T, t_1, t_2, t_3, t_4)} \mid \mathbf{t}_1 \,\&_\mathbf{T}\, \mathbf{t}_2}$$
$$\boxed{\mathbf{h} +::= \lambda \mid \mathbf{C^D} \mid \mathbf{refl} \mid =}$$

$$\frac{\Gamma \vdash t \rightharpoonup \mathbf{t} \Rightarrow \mathbf{T}}{\Gamma \vdash \mathit{refl}_t \rightharpoonup \mathbf{refl(t)}_{\vdash \mathbf{t} \cong \mathbf{t}} \Rightarrow \mathbf{t} =_\mathbf{T} \mathbf{t}} \quad \textsc{ElabRefl}$$

$$\frac{\Gamma \vdash \mathbf{t_w} \Rightarrow \mathbf{T} \qquad \Gamma \vdash \mathbf{t}_1 \Leftarrow \mathbf{T} \qquad \Gamma \vdash \mathbf{t}_2 \Leftarrow \mathbf{T} \quad \Gamma|\Gamma \vdash \mathbf{t_w} \sqsubseteq^{\leftarrow}_{\rightarrow} \mathbf{t}_1 \qquad \Gamma|\Gamma \vdash \mathbf{t_w} \sqsubseteq^{\leftarrow}_{\rightarrow} \mathbf{t}_2}{\Gamma \vdash \mathbf{refl(t_w)}_{\vdash \mathbf{t}_1 \cong \mathbf{t}_2} \Rightarrow \mathbf{t}_1 =_\mathbf{T} \mathbf{t}_2} \quad \textsc{CastRefl}$$

$$\frac{\Gamma \vdash \mathbf{t}_1 \Leftarrow \mathbf{T} \qquad \Gamma \vdash \mathbf{t}_2 \Leftarrow \mathbf{T}}{\Gamma \vdash \mathbf{t}_1 \,\&_\mathbf{T}\, \mathbf{t}_2 \Rightarrow \mathbf{T}} \quad \textsc{CastComp}$$

Fig. 7. CastEq: Syntax, Key Typing and Elaboration Rules

$\mathbf{J}$ takes a type T_P parameterized over a value of type T, [4] along with two values of type T. Then, given a value of type $[t_1/x]T_P$, and a proof t_{eq} that t_1 and t_2 are equal, the elimination has type $[t_2/x]T_P$. That is, if two values are equal, we can take any term whose type refers to the first, and transform it into a term whose type refers to the second.

4.2 CastEq Syntax and Typing

Fig. 7 extends CastCIC to CastEq by adding propositional equality and the gradual composition operator. We extend the syntax for a static head $\mathbf{h}$ to include value constructors, not just types, which is useful when defining the semantics of composition. A proof of reflexivity is written as $\mathbf{refl(t_w)}_{\vdash \mathbf{t}_1 \cong \mathbf{t}_2}$, where $\mathbf{t}_1$ and $\mathbf{t}_2$ are the equated terms, and $\mathbf{t_w}$ is a witness of the (dynamic) consistency of those endpoints. We borrow the notation $(\mathbf{t_w})_{\vdash \mathbf{t}_1 \cong \mathbf{t}_2}$ from Garcia et al. [2016] to indicate that $\mathbf{t_w}$ contains information supporting the (dynamic) consistency of $\mathbf{t}_1$ and $\mathbf{t}_2$. Composition is ascribed with the type of its arguments so that we can ascribe the proper $\mathbf{T}$ when the composition of two terms steps to $\mho_\mathbf{T}$.

For typing, CastComp synthesizes a composition's ascribed type when both arguments check against that type. In CastRefl, $\mathbf{refl(t_w)}_{\vdash \mathbf{t}_1 \cong \mathbf{t}_2}$ synthesizes $\mathbf{t}_1 =_\mathbf{T} \mathbf{t}_2$ if the witness $\mathbf{t_w}$ is as precise as both $\mathbf{t}_1$ and $\mathbf{t}_2$. In ElabRefl, refl_t is elaborated into $\mathbf{refl(t)}_{\vdash \mathbf{t} \cong \mathbf{t}}$, i.e., a term serves as the initial witness that it is equal to itself. If t is imprecise, casts applied to the equality proof may produce more precise witnesses, but the programmer never constructs a witness directly. We omit typing rules for $=_\mathbf{T}$ and $\mathbf{J}$, as they mirror the lemmas in Fig. 6, as do their elaboration rules.

Precision must be closed under convertibility because, as Lennon-Bertrand et al. [2022] note, syntactic precision is not preserved by stepping the less precise term. Since $?_\mathbf{T}$ is less precise than $\mathbf{x}$, the less precise term may reduce in a way that is blocked for the other term. So for contextual steps to preserve CastRefl, the results of stepping related terms must be related.

RedJ
$$\mathbf{J}(\mathbf{x}.\mathbf{T_P}, \mathbf{t}_1, \mathbf{t}_2, \mathbf{t_{P1}}, \mathbf{refl}(\mathbf{t_w})_{\vdash \mathbf{t}'_1 \cong \mathbf{t}'_2}) \rightsquigarrow \langle [\mathbf{t}_2/\mathbf{x}]\mathbf{T_P} \Leftarrow [\mathbf{t_w}/\mathbf{x}]\mathbf{T_P}\rangle\langle [\mathbf{t_w}/\mathbf{x}]\mathbf{T_P} \Leftarrow [\mathbf{t}_1/\mathbf{x}]\mathbf{T_P}\rangle \mathbf{t_{P1}}$$

RedEqGerm
$$\frac{\mathbf{t}_1 =_{\mathbf{T}} \mathbf{t}_2 \neq \mathsf{germ}_\ell(=)}{\langle ?_{\mathbf{Type}_\ell} \Leftarrow \mathbf{t}_1 =_{\mathbf{T}} \mathbf{t}_2\rangle \mathbf{t} \rightsquigarrow \langle ?_{\mathbf{Type}_\ell} \Leftarrow ?_{?_{\mathbf{Type}_\ell}} =_{?_{\mathbf{Type}_\ell}} ?_{?_{\mathbf{Type}_\ell}}\rangle\langle ?_{?_{\mathbf{Type}_\ell}} =_{?_{\mathbf{Type}_\ell}} ?_{?_{\mathbf{Type}_\ell}} \Leftarrow \mathbf{t}_1 =_{\mathbf{T}} \mathbf{t}_2\rangle \mathbf{t}}$$

PropEqUnk
$$?_{\mathbf{t}_1 =_{\mathbf{T}} \mathbf{t}_2} \rightsquigarrow \mathbf{refl}(\mathbf{t}_1 \,\&_{\mathbf{T}}\, \mathbf{t}_2)_{\vdash \mathbf{t}_1 \cong \mathbf{t}_2}$$

PropEq(Unk,Err)
$$\mho_{\mathbf{t}_1 =_{\mathbf{T}} \mathbf{t}_2} \rightsquigarrow \mathbf{refl}(\mho_{\mathbf{T}})_{\vdash \mathbf{t}_1 \cong \mathbf{t}_2}$$

RedCastEq
$$\langle \mathbf{t}'_1 =_{\mathbf{T}'} \mathbf{t}'_2 \Leftarrow \mathbf{t}_1 =_{\mathbf{T}} \mathbf{t}_2\rangle \mathbf{refl}(\mathbf{t_w})_{\vdash \mathbf{t}''_1 \cong \mathbf{t}''_2} \rightsquigarrow \mathbf{refl}((\langle \mathbf{T}' \Leftarrow \mathbf{T}\rangle \mathbf{t_w} \,\&_{\mathbf{T}'}\, \mathbf{t}'_1 \,\&_{\mathbf{T}'}\, \mathbf{t}'_2))_{\vdash \mathbf{t}'_1 \cong \mathbf{t}'_2}$$

Fig. 8. CastEq: Reductions for Casts and J

4.3 Cast Semantics

A challenge with gradual equality is designing its dynamic semantics. In a fully static language, refl always equates identical values, so J performs no computation other than pattern matching on the proof of equality. In the presence of type imprecision, **J** must perform casts. We also need reductions for casts between equality types. Figure 8 gives reductions for **J** and casts. The RedJ rule reduces by casting *through* the motive $\mathbf{T_P}$ with $\mathbf{x}$ bound to the witness $\mathbf{t_w}$. The typing of equality guarantees that this witness $\mathbf{t_w}$ is as precise as either $\mathbf{t}_1$ and $\mathbf{t}_2$. So $[\mathbf{t_w}/\mathbf{x}]\mathbf{T_P}$ is like a middle type, since it is more precise than $[\mathbf{t}_1/\mathbf{x}]\mathbf{T_P}$ (the type of $\mathbf{t_{P1}}$) and $[\mathbf{t}_2/\mathbf{x}]\mathbf{T_P}$ (the type of the result).

Why cast through the middle, and not directly from $[\mathbf{t}_1/\mathbf{x}]\mathbf{T_P}$ to $[\mathbf{t}_2/\mathbf{x}]\mathbf{T_P}$? As §2.3 showed, the witness tracks constraints as the program runs, and since composition is monotone, its precision only increases. So constraints are remembered, and **J** only succeeds if the equated terms are consistent with all those remembered contraints, allowing the programmer to see when a static constraint has been dynamically violated. Also, the witness ensures that equalities between inconsistent values cannot be used without flagging an error. Without a witness, one could have $?_{2=_{\mathbb{N}}5}$, despite the type being statically uninhabited. Then **J** could use this equality to convert from **Vec Float** (2 mod 3) to **Vec Float** (5 mod 3): the cast would succeed, despite the absurdity of the initial equality. Going through the middle type catches such absurd cases.

For casting **refl** between equality types, the RedCastEq rule first casts the witness to the correct type. The typing rule CastRefl requires the witness to be as precise as the endpoints, but the result of casting the witness might not fulfill this! So the witness is composed with both endpoints, producing a precision-related result. These casts are precisely why we need a composition operator.

The propagation rules PropEqUnk and PropEqErr reduce **?** and $\mho$ at equality types to **refl** with the least and most precise witnesses, respectively. RedEqGerm casts an equality proof to $?_{\mathbf{Type}_\ell}$ by casting through the germ type, just like with functions and constructors.

4.4 Semantics of Composition

Figure 9 gives the semantics of composition. Technically, we do not need composition as an operator in CastEq itself, but only for witnesses and cast type ascriptions. However, because dependent types remove the separation between terms and types, witnesses and cast types need dynamic semantics. So for simplicity, we let witnesses and cast-types be any CastEq terms, and add composition to CastEq's semantics, rather than duplicating CastEq's semantics for a witness-specific language.

Several composition rules receive two or three different sets of type ascriptions, with one on the composition itself, and possibly one on each of the composed terms. In such cases, the choice

[4]The full J in type theory parameterizes $\mathbf{T_P}$ over the equality proof. Section 7.1.2 shows why this is not needed for GEq.

$\boxed{\text{UnkVal } \mathbf{T} \textit{ (Types with ? and } \mathcal{U} \textit{ forms)}}$

$$\frac{}{\text{UnkVal } \mathbf{Type}_\ell} \qquad \frac{}{\text{UnkVal } \mathbf{C}(\overline{\mathbf{t}})} \qquad \frac{}{\text{UnkVal } ?_{\mathbf{Type}_\ell}}$$

$\boxed{\mathbf{t}_1 \rightsquigarrow \mathbf{t}_2 \textit{ (Composition reductions)}}$

$$\textsc{RedCompUnkL}\ \frac{\text{UnkVal } \mathbf{T}}{?_{\mathbf{T}} \mathbin{\&_{\mathbf{T}'}} \mathbf{t} \rightsquigarrow \mathbf{t}} \qquad \textsc{RedCompUnkR}\ \frac{\text{UnkVal } \mathbf{T}}{\mathbf{t} \mathbin{\&_{\mathbf{T}'}} ?_{\mathbf{T}} \rightsquigarrow \mathbf{t}} \qquad \textsc{RedCompErrL}\ \frac{\text{UnkVal } \mathbf{T}}{\mathcal{U}_{\mathbf{T}} \mathbin{\&_{\mathbf{T}'}} \mathbf{t} \rightsquigarrow \mathcal{U}_{\mathbf{T}}}$$

$$\textsc{RedCompErrR}\ \frac{\text{UnkVal } \mathbf{T}}{\mathbf{t} \mathbin{\&_{\mathbf{T}'}} \mathcal{U}_{\mathbf{T}} \rightsquigarrow \mathcal{U}_{\mathbf{T}}}$$

$$\textsc{RedCompGerm}\ (\langle ?_{\mathbf{Type}_\ell} \Leftarrow \mathrm{germ}(\mathbf{h})\rangle \mathbf{t}_1) \mathbin{\&_{?_{\mathbf{Type}_\ell}}} (\langle ?_{\mathbf{Type}_\ell} \Leftarrow \mathrm{germ}(\mathbf{h})\rangle \mathbf{t}_2) \rightsquigarrow \langle ?_{\mathbf{Type}_\ell} \Leftarrow \mathrm{germ}(\mathbf{h})\rangle (\mathbf{t}_1 \mathbin{\&_{\mathrm{germ}(\mathbf{h})}} \mathbf{t}_2)$$

$$\textsc{RedCompGermErr}\ \frac{\mathbf{h}_1 \neq \mathbf{h}_2}{(\langle ?_{\mathbf{Type}_\ell} \Leftarrow \mathrm{germ}(\mathbf{h}_1)\rangle \mathbf{t}_1) \mathbin{\&_{?_{\mathbf{Type}_\ell}}} (\langle ?_{\mathbf{Type}_\ell} \Leftarrow \mathrm{germ}(\mathbf{h}_2)\rangle \mathbf{t}_2) \rightsquigarrow \mathcal{U}_{?_{\mathbf{Type}_\ell}}}$$

$$\textsc{RedCompHeadErr}\ \frac{\mathrm{head}(\mathbf{t}_1) = \mathbf{h}_1 \quad \mathrm{head}(\mathbf{t}_2) = \mathbf{h}_2 \quad \mathbf{h}_1 \neq \mathbf{h}_2}{\mathbf{t}_1 \mathbin{\&_{\mathbf{T}}} \mathbf{t}_2 \rightsquigarrow \mathcal{U}_{\mathbf{T}}}$$

$$\textsc{RedCompEq}\ \frac{\mathbf{t}_1'' := \langle \mathbf{T}_1 \mathbin{\&_{\mathbf{Type}_\ell}} \mathbf{T}_2 \Leftarrow \mathbf{T}_1\rangle \mathbf{t}_1 \mathbin{\&_{(\mathbf{T}_1 \&_{\mathbf{Type}_\ell} \mathbf{T}_2)}} \langle \mathbf{T}_1 \mathbin{\&_{\mathbf{Type}_\ell}} \mathbf{T}_2 \Leftarrow \mathbf{T}_2\rangle \mathbf{t}_2 \quad \mathbf{t}_2'' := \langle \mathbf{T}_1 \mathbin{\&_{\mathbf{Type}_\ell}} \mathbf{T}_2 \Leftarrow \mathbf{T}_1\rangle \mathbf{t}_1' \mathbin{\&_{(\mathbf{T}_1 \&_{\mathbf{Type}_\ell} \mathbf{T}_2)}} \langle \mathbf{T}_1 \mathbin{\&_{\mathbf{Type}_\ell}} \mathbf{T}_2 \Leftarrow \mathbf{T}_2\rangle \mathbf{t}_2'}{(\mathbf{t}_1 =_{\mathbf{T}_1} \mathbf{t}_1') \mathbin{\&_{\mathbf{Type}_\ell}} (\mathbf{t}_2 =_{\mathbf{T}_2} \mathbf{t}_2') \rightsquigarrow \mathbf{t}_1'' =_{(\mathbf{T}_1 \&_{\mathbf{Type}_\ell} \mathbf{T}_2)} \mathbf{t}_2''}$$

$$\textsc{RedCompRefl}\ \mathbf{refl}(\mathbf{t}_1)_{\vdash \mathbf{t}_L \cong \mathbf{t}_R} \mathbin{\&_{\mathbf{t}_L'' =_{\mathbf{T}} \mathbf{t}_R''}} \mathbf{refl}(\mathbf{t}_2)_{\vdash \mathbf{t}_L' \cong \mathbf{t}_R'} \rightsquigarrow \mathbf{refl}(\mathbf{t}_1 \mathbin{\&_{\mathbf{T}}} \mathbf{t}_2)_{\vdash \mathbf{t}_L \cong \mathbf{t}_R}$$

$$\textsc{RedCompLam}\ \mathbf{t}_1 \mathbin{\&_{(\mathbf{x}:\mathbf{T}_1) \to \mathbf{T}_2}} \mathbf{t}_2 \rightsquigarrow \lambda(\mathbf{x} : \mathbf{T}_1).\ ((\mathbf{t}_1\ \mathbf{x}) \mathbin{\&_{\mathbf{T}_2}} (\mathbf{t}_2\ \mathbf{x}))$$

$$\textsc{RedCompInd}\ \mathbf{C}(\overline{\mathbf{t}_1}) \mathbin{\&_{\mathbf{Type}_\ell}} \mathbf{C}(\overline{\mathbf{t}_2}) \rightsquigarrow \mathbf{C}(\overline{\&}(\mathbf{Params}(\mathbf{C}, \mathbf{i}), \overline{\mathbf{t}_1}, \overline{\mathbf{t}_2}))$$

$$\textsc{RedCompCon}\ \mathbf{D}^{\mathbf{C}}(\overline{\mathbf{t}_1}, \overline{\mathbf{t}_2}) \mathbin{\&_{\mathbf{C}(\overline{\mathbf{t}'})}} \mathbf{D}^{\mathbf{C}}(\overline{\mathbf{t}_1''}, \overline{\mathbf{t}_2''}) \rightsquigarrow \mathbf{D}^{\mathbf{C}}(\overline{\mathbf{t}}, \overline{\&}(\mathbf{Args}(\mathbf{C}, \mathbf{i}, \mathbf{D}_k), \overline{\mathbf{t}_2}, \overline{\mathbf{t}_2''}))$$

$$\textsc{RedCompPi}\ (\mathbf{x} : \mathbf{T}_1) \to \mathbf{T}_1' \mathbin{\&_{\mathbf{Type}_\ell}} (\mathbf{x} : \mathbf{T}_2) \to \mathbf{T}_2' \rightsquigarrow (\mathbf{x} : (\mathbf{T}_1 \mathbin{\&_{\mathbf{Type}_\ell}} \mathbf{T}_2)) \to [\langle \mathbf{T}_1 \Leftarrow \mathbf{T}_1 \mathbin{\&_{\mathbf{Type}_\ell}} \mathbf{T}_2\rangle \mathbf{x}/\mathbf{x}]\mathbf{T}_1' \mathbin{\&_{\mathbf{Type}_\ell}} [\langle \mathbf{T}_2 \Leftarrow \mathbf{T}_1 \mathbin{\&_{\mathbf{Type}_\ell}} \mathbf{T}_2\rangle \mathbf{x}/\mathbf{x}]\mathbf{T}_2'$$

$\boxed{\overline{\&}(\overline{(\mathbf{x} : \mathbf{T})}, \overline{\mathbf{t}_1}, \overline{\mathbf{t}_2}) \textit{ (Telescope Composition)}}$

$$\overline{\&}(\cdot, \cdot, \cdot) := \cdot \qquad \overline{\&}((\mathbf{x} : \mathbf{T}_1) \cdot \overline{(\mathbf{y} : \mathbf{T}_2)}, \mathbf{t}_1 \cdot \overline{\mathbf{t}_1'}, \mathbf{t}_2 \cdot \overline{\mathbf{t}_2'}) := (\mathbf{t}_1 \mathbin{\&_{\mathbf{T}_1}} \mathbf{t}_2) \cdot \overline{\&}(\overline{(\mathbf{y} : [(\mathbf{t}_1 \mathbin{\&_{\mathbf{T}_1}} \mathbf{t}_2)/\mathbf{x}]\mathbf{T}_2)}, seq_1, seq_2) \ \textit{where}$$

$$seq_1 := \overline{\langle [(\mathbf{t}_1 \mathbin{\&_{\mathbf{T}_1}} \mathbf{t}_2)/\mathbf{x}]\mathbf{T}_2 \Leftarrow [\mathbf{t}_1/\mathbf{x}]\mathbf{T}_2\rangle \mathbf{t}_1'} \qquad seq_2 := \overline{\langle [(\mathbf{t}_1 \mathbin{\&_{\mathbf{T}_1}} \mathbf{t}_2)/\mathbf{x}]\mathbf{T}_2 \Leftarrow [\mathbf{t}_2/\mathbf{x}]\mathbf{T}_2\rangle \mathbf{t}_2'}$$

Fig. 9. CastEq: Semantics for &

of ascription in the reduct is arbitrary. For example, RedCompErrL uses the ascription from $\mathcal{U}$, rather than what was on &. While we do not require that the ascriptions be syntactically equal, the typing rules ensure that syntactically distinct ascriptions are definitionally equal, so the choice of ascription in the reduct does not affect the final result of evaluation.

Each composition rule resembles a unification rule, where each use of ? is treated as a unification variable. Terms that have different head tags compose to an error, and terms that have the same head tag compose by composing their parts. Any place where one term has ? but the other term contains more precise information, the precise information is retained in the output. This ensures

that when $\mathbf{t}_1 \mathbin{\&_T} \mathbf{t}_2$ reduces, the result is a term that is as precise as both $\mathbf{t}_1$ and $\mathbf{t}_2$. For $?_T \mathbin{\&_T} \mathbf{t}$, we produce $\mathbf{t}$ (RedCompUnk(L,R)), since $\mathbf{t}$ is always as precise as itself and $?_T$. Likewise, the rules that produce $\mathcal{U}$ satisfy this, since it is the most precise term. We see this in RedCompUnk(L,R) , which composes with $\mathcal{U}$, and in RedCompHeadErr and RedCompGermErr, where composing non-neutral terms with distinct heads reduces to $\mathcal{U}$.

The remaining rules compose terms with the same head $\mathbf{h}$, such as when both are functions or both are built with the same $\mathbf{D}^C$. In these cases, the head $\mathbf{h}$ is applied to the respective composition of the arguments, e.g., the composition of functions is a function returning the composition of the bodies. For equality proofs and and inhabitants of $?_{\mathbf{Type}_\ell}$, the head can be applied directly (RedCompLam, RedCompRefl, RedCompGerm). In RedCompLam, composing functions yields a new function that composes the results of the given functions for each argument. In the remaining cases, we must account for how types of later arguments depend on the values of earlier arguments. RedCompPi produces a domain by composing the argument domains, which is the type of the parameter $\mathbf{x}$. The codomain $\mathbf{x}$'s each have their own domain types, so we cast all uses of $\mathbf{x}$ from the composed type to the expected type. RedCompEq composes equality types: the type ascriptions are composed, then the equated terms are cast to this composed type. Composing the results of these casts yields the result endpoints.

The most complex rules are RedCompInd and RedCompCon. Because type and data constructors have dependent function types, their arguments are *telescopes:* the type of later arguments may depend on the *values* of previous parameters and arguments. To compose type or data constructor applications, we compose the parameters and arguments element-wise, but composing two arguments changes the type of later arguments. To compose telescopes, the metafunction $\overline{\&}$ traverses the types of type and data constructors, composing arguments element-wise and adding casts to the bound variables in later arguments.

To see why composing needs casts, consider dependent pairs formulated as inductive types.

$$\begin{aligned}&\mathbf{data\ DPair}\ :\ (\mathbf{X} : \mathbf{Type}) \to (\mathbf{P} : \mathbf{X} \to \mathbf{Type}) \to \mathbf{Type\ where}\\ &\quad \mathbf{mkDPair} : (\mathbf{x} : \mathbf{X}) \to \mathbf{P\ x} \to \mathbf{DPair\ X\ P}\end{aligned}$$

One example of a dependent pair type is $\mathbf{DPair}\ (\mathbb{N} \times \mathbb{N})\ (\lambda \mathbf{x}.\ (\pi_1\ \mathbf{x}) + (\pi_2\ \mathbf{x}) =_{\mathbb{N}} 3)$, i.e., the Curry-Howard equivalent of "there exists a pair of numbers such that adding them yields 3." Suppose we want to compose two inhabitants of this type, say $\mathbf{mkDPair}\ (1, ?_{\mathbb{N}})\ \mathbf{refl}(3)_{\vdash 1+?_{\mathbb{N}} \cong 3}$ and $\mathbf{mkDPair}\ (?_{\mathbb{N}}, 2)\ \mathbf{refl}(3)_{\vdash ?_{\mathbb{N}}+2 \cong 3}$. To compose the first element, we can produce $(1 \mathbin{\&_{\mathbb{N}}} ?_{\mathbb{N}}, ?_{\mathbb{N}} \mathbin{\&_{\mathbb{N}}} 2)$, which reduces to $(1, 2)$. However, for the second element, the two proofs $\mathbf{refl}(3)_{\vdash 1+?_{\mathbb{N}} \cong 3}$ and $\mathbf{refl}(3)_{\vdash ?_{\mathbb{N}}+2 \cong 3}$ do not have the same type: they equate different terms, so we cannot compose them! Instead, we must first cast each to type $(1 \mathbin{\&_{\mathbb{N}}} ?_{\mathbb{N}}) + (?_{\mathbb{N}} \mathbin{\&_{\mathbb{N}}} 2) =_{\mathbb{N}} 3$, i.e., the value obtained by replacing $\mathbf{x}$ with the composition of the pairs' first elements in the term $((\pi_1\ \mathbf{x}) + (\pi_2\ \mathbf{x}) =_{\mathbb{N}} 3)$. This gives a final result of $\mathbf{mkDPair}\ (1, 2)\ (\mathbf{refl}(3)_{\vdash 3 \cong 3})$.

5 PARAMETERIZED METATHEORY: CRITERIA FOR PRECISION AND CONSISTENCY

GEq is now defined except for CastEq's precision and consistency relations. For non-dependent languages, the semantics of precision can be justified either in terms of sets of static terms [Garcia et al. 2016] or in terms of semantic precision [New and Ahmed 2018]. Such justifications are difficult with dependent types. Our approach is different: we define the important metatheoretic criteria for GEq without referring to precision and consistency, then describe the criteria precision and consistency must fulfill to prove the desired metatheoretic properties. Meeting these criteria guides and justifies our definition of precision and consistency (§6). We see precision and consistency as a means to the end of the desired metatheory.

5.1 Stating the Gradual Guarantees

The gradual guarantees state that reducing the precision of a surface term introduces no new static or dynamic errors. However, to state them formally, we must define what precision means for surface terms. We follow Lennon-Bertrand et al. [2022] and define surface precision as the relation generated by $t \sqsubseteq_{\mathsf{Surf}} ?_{@i}$, plus all the usual structural rules. Essentially, $t \sqsubseteq_{\mathsf{Surf}} t'$ holds if we can obtain t' by replacing some parts of t with some $?_{@i}$. To guarantee preservation of typing, we also need such replacements to be *universe adequate* [Lennon-Bertrand et al. 2022]. We say that the judgment $t \sqsubseteq_{\mathsf{Surf}} t'$ is universe adequate if, for every subterm r of t, when $\Gamma \vdash r \rightarrowtriangle \mathbf{r} \Rightarrow \mathbf{T}$ and $\Gamma \vdash \mathbf{T} \Rightarrow_{\mathbf{Type}} \mathbf{Type_i}$, then any uses of $r \sqsubseteq_{\mathsf{Surf}} ?_{@j}$ have $i = j$. This essentially says t is t' with some subterms replaced by $?_{@i}$ *for the right* i. We can now state the static gradual guarantee:

Definition 5.1 (Static Gradual Guarantee). If $\cdot \vdash t : T$ and $t \sqsubseteq_{\mathsf{Surf}} t'$ universe-adequately, then $\cdot \vdash t' : T$.

That is, reducing the precision of a program causes no new type errors.

To state the dynamic guarantee without referring to CastEq precision, we must formalize what it means to introduce no new dynamic errors. We follow New and Ahmed [2018] and do this with *semantic precision*, which compares terms by quantifying over all possible boolean contexts. We use booleans because of their simplicity: gradual booleans have only the four values $\mathbf{true}$, $\mathbf{false}$, $?_{\mathbb{B}}$ and $\mathcal{U}_{\mathbb{B}}$. If a context exists such that reducing a term's precision changes the result from $\mathbf{true}$ to $\mathbf{false}$, then we have violated the guarantee that precision only affects behavior via errors. Likewise, if a context exists such that reducing precision turns $\mathbf{true}$ to $\mathcal{U}_{\mathbb{B}}$, then reducing precision introduced a new error. By defining semantic precision in terms of *all* contexts, we capture the idea that the above behaviors are impossible for precision-related terms. We formalize this as follows:

Definition 5.2 (Semantic Precision). *Boolean precision* $\sqsubseteq_{\mathbb{B}}$ is defined by $\mathbf{true} \sqsubseteq_{\mathbb{B}} \mathbf{true}$, $\mathbf{false} \sqsubseteq_{\mathbb{B}} \mathbf{false}$, $\mathcal{U}_{\mathbb{B}} \sqsubseteq_{\mathbb{B}} \mathbf{b}$, and $\mathbf{b} \sqsubseteq_{\mathbb{B}} ?_{\mathbb{B}}$ for all $\mathbf{b} : \mathbb{B}$. Then two closed terms are related by *semantic precision*, written $\vDash \mathbf{t} \sqsubseteq^{\vDash} \mathbf{t'} : \mathbf{T}$ if, for all $C : \mathbf{T} \to \mathbb{B}$, whenever $C[\mathbf{t}] \longrightarrow^* \mathbf{b}$, then $C[\mathbf{t'}] \longrightarrow^* \mathbf{b'}$ and $\mathbf{b} \sqsubseteq_{\mathbb{B}} \mathbf{b'}$.

Then the gradual guarantee states that reducing a surface term's precision causes a corresponding reduction in the semantic precision of the surface terms' elaborations.

Definition 5.3 (Dynamic Gradual Guarantee). Suppose $\cdot \vdash t \rightarrowtriangle \mathbf{t} \Leftarrow \mathbf{T}$ and $\cdot \vdash t' \rightarrowtriangle \mathbf{t'} \Leftarrow \mathbf{T}$. If $t \sqsubseteq_{\mathsf{Surf}} t'$ universe-adequately, then $\Gamma \vDash \mathbf{t} \sqsubseteq^{\vDash} \mathbf{t'}$.

5.2 Necessary Properties of Precision and Consistency

Next, we list properties that, if satisfied by $\sqsubseteq^{\longleftarrow}_{\longrightarrow}$ and $\cong_{\longrightarrow}$, suffice to prove type safety, conservative extension of CIC, well-typedness of elaboration, and the gradual guarantees. Each criterion is accompanied by a specific case of the safety or gradual guarantee proofs that motivates its inclusion. We also include criteria that composition should satisfy. While the semantics appear in a prior section (§4.4), the criteria are new from GCIC, so we list them to highlight our contribution.

Though $\sqsubseteq^{\longleftarrow}_{\longrightarrow}$ is ideal for typing witnesses, it is too lenient to express the monotonicity properties of CastEq. In particular, if the monotonicity of reduction is phrased with $\sqsubseteq^{\longleftarrow}_{\longrightarrow}$, then consistency must also be closed under convertibility, which would make it undecidable even when its operands terminate. So we introduce a strictly stronger relation $\sqsubseteq_{\longrightarrow}$ which, like $\cong_{\longrightarrow}$, only compares after reductions, and not before. This distinction is acceptable because the precision side-condition of CastRefl is never used to prove safety or monotonicity. Rather, the side-condition ensures that witnesses always entail at least as much information as the equated terms. Including the side-condition in CastEq's type conveniently captures the invariant that, when refl_t is elaborated with initial witness $\mathbf{t}$, future witnesses are never lose the information from $\mathbf{t}$.

Safety and Elaboration. For elaboration to preserve types, precision must be reflexive so the initial witness for *refl* is valid. For safety, progress requires that each well-typed non-value can step. So composition must step for each non-value. For preservation, each reduction must preserve types, including composition reductions. If composition yields a precision lower-bound and precision is transitive, the side-conditions of CastRefl can be preserved.

Lemma 5.4 (Precision Reflexive). *If* $\Gamma \vdash \mathbf{t} \Leftarrow \mathbf{T}$ *then* $\Gamma|\Gamma \vdash \mathbf{t} \sqsubseteq_{\longrightarrow} \mathbf{t}$ *(For ElabRefl to produce an elaboration that satisfies the* $\sqsubseteq_{\longrightarrow}$ *side-condition of CastRefl).*

Lemma 5.5 (Composition Safety). *If* $\mathbf{t}_1 \&_{\mathbf{T}} \mathbf{t}_2$ *is not a value and* $\Gamma \vdash \mathbf{t}_1 \&_{\mathbf{T}} \mathbf{t}_2 \Leftarrow \mathbf{T}$*, then* $\mathbf{t}_1 \&_{\mathbf{T}} \mathbf{t}_2 \longrightarrow \mathbf{t}_3$ *for some* $\mathbf{t}_3$ *and* $\Gamma \vdash \mathbf{t}_3 \Leftarrow \mathbf{T}$ *(For progress and preservation)*

Lemma 5.6 (Composition Confluence). *If* $\mathbf{t}_1 \&_{\mathbf{T}} \mathbf{t}_2 \Rrightarrow \mathbf{t}_3$ *and* $\mathbf{t}_1 \&_{\mathbf{T}} \mathbf{t}_2 \Rrightarrow \mathbf{t}'_3$ *maximally, then* $\mathbf{t}_3 \Rrightarrow \mathbf{t}'_3$*, where* $\Rrightarrow$ *is the parallel reduction relation, standard in confluence proofs [Takahashi 1995] (For confluence, which is needed to show that* β*-reductions preserve types);*

Lemma 5.7 (Composition Lower Bound). *If* $\Gamma \vdash \mathbf{t}_1 \&_{\mathbf{T}} \mathbf{t}_2 \Leftarrow \mathbf{T}$*, then* $\Gamma|\Gamma \vdash \mathbf{t}_1 \&_{\mathbf{T}} \mathbf{t}_2 \sqsubseteq_{\longrightarrow} \mathbf{t}_1$ *and* $\Gamma|\Gamma \vdash \mathbf{t}_1 \&_{\mathbf{T}} \mathbf{t}_2 \sqsubseteq_{\longrightarrow} \mathbf{t}_2$ *(Preserving the* $\sqsubseteq^{\longleftarrow}_{\longrightarrow}$ *condition of CastRefl for reduction RedCastEq);*

Lemma 5.8 (Precision Transitive). *If* $\Gamma_1|\Gamma_2 \vdash \mathbf{t}_1 \sqsubseteq_{\longrightarrow} \mathbf{t}_2$ *and* $\Gamma_2|\Gamma_3 \vdash \mathbf{t}_2 \sqsubseteq_{\longrightarrow} \mathbf{t}_3 : \mathbf{T}$ *then* $\Gamma_1|\Gamma_3 \vdash \mathbf{t}_1 \sqsubseteq_{\longrightarrow} \mathbf{t}_3$ *(Preserving the* $\sqsubseteq^{\longleftarrow}_{\longrightarrow}$ *side-condition of CastRefl for reduction RedCastEq);*

Lemma 5.9 (Precision Modulo Conversion). *If* $\Gamma_1|\Gamma_2 \vdash \mathbf{t}_1 \sqsubseteq^{\longleftarrow}_{\longrightarrow} \mathbf{t}_2$*, where* $\mathbf{t}_2 \longrightarrow^* \mathbf{t}'_2$*, then* $\Gamma_1|\Gamma_2 \vdash \mathbf{t}_1 \sqsubseteq^{\longleftarrow}_{\longrightarrow} \mathbf{t}'_2$ *(Preservation of CastRefl under contextual reduction)*

Conservativity. If GEq is to conservatively extend CIC, then a fully-static program should be well-typed in CIC if and only if it is well-typed in GEq. For the most part, the rules only differ when $?$ is involved, but the major exception is ElabCst, which let us replace a type with any consistent type (after conversion). So for fully static terms, consistency should coincide with syntactic equality.

Lemma 5.10 (Static Consistency). *For any static terms* t_1 *and* t_2*, let* $\mathbf{t}_1$ *and* $\mathbf{t}_2$ *be their embedding in* CastEq*. Then* $\mathbf{t}_1 \cong_{\longrightarrow} \mathbf{t}_2$ *iff* $\mathrm{t}_1 =_{\alpha\beta} \mathrm{t}_2$*, i.e., if they are statically definitionally equal (For* GEq *to conservatively extend CIC).*

Monotonicity. The last group of properties relate to the gradual guarantees. The dynamic gradual guarantee requires that evaluating precision-related terms produces precision-related results. Because of the dependency in dependent types, proving the static guarantee relies on the proof of the dynamic guarantee: ElabCst reduces types before comparing for consistency, so precision of types before reduction should be preserved, and reducing the precision of a type should make it consistent with no fewer types. Likewise, to show the static guarantee, elaboration must be monotone in both synthesized types and elaborated terms, since dependent application uses the argument's elaboration in the return type.

Lemma 5.11 (Cast Monotonicity). *Suppose that* $\Gamma_1|\Gamma_2 \vdash \mathbf{t}_1 \sqsubseteq_{\longrightarrow} \mathbf{t}_2$*,* $\Gamma_1 \vdash \mathbf{t}_1 \Rrightarrow \mathbf{T}_1$ *and* $\Gamma_2 \vdash \mathbf{t}_2 \Rrightarrow \mathbf{T}_2$ *where* $\Gamma_1|\Gamma_1 \vdash \mathbf{T}_1 \sqsubseteq_{\longrightarrow} \mathbf{T}'_1$ *and* $\Gamma_2|\Gamma_2 \vdash \mathbf{T}_2 \sqsubseteq_{\longrightarrow} \mathbf{T}'_2$*. Then* $\Gamma_1|\Gamma_2 \vdash \langle \mathbf{T}'_1 \Leftarrow \mathbf{T}_1 \rangle \mathbf{t}_1 \sqsubseteq_{\longrightarrow} \langle \mathbf{T}'_2 \Leftarrow \mathbf{T}_2 \rangle \mathbf{t}_2$ *(For ElabCst to produce* $\sqsubseteq_{\longrightarrow}$*-related elaborations for* $\sqsubseteq_{\mathsf{Surf}}$*-related inputs)*

Lemma 5.12 (Substitution Monotone). *Suppose* $\Gamma_1|\Gamma_2 \vdash \mathbf{t}_1 \sqsubseteq_{\longrightarrow} \mathbf{t}_2$*, where* $\Gamma_1 \vdash \mathbf{t}_1 \Rrightarrow \mathbf{T}_1$ *and* $\Gamma_2 \vdash \mathbf{t}_2 \Rrightarrow \mathbf{T}_2$*. If* $\Gamma_1(\mathbf{x} : \mathbf{T}_1)\Delta_1|\Gamma_2(\mathbf{x} : \mathbf{T}_2)\Delta_2 \vdash \mathbf{t}'_1 \sqsubseteq_{\longrightarrow} \mathbf{t}'_2$*,*
then $\Gamma_1[\mathbf{t}_1/\mathbf{x}]\Delta_1|\Gamma_2[\mathbf{t}_2/\mathbf{x}]\Delta_2 \vdash [\mathbf{t}_1/\mathbf{x}]\mathbf{t}'_1 \sqsubseteq_{\longrightarrow} [\mathbf{t}_2/\mathbf{x}]\mathbf{t}'_2$ *(For ElabApp to be monotone in the return type)*

Lemma 5.13 (Reduction Monotone). *If* $\Gamma_1|\Gamma_2 \vdash \mathbf{t}_1 \sqsubseteq_{\longrightarrow} \mathbf{t}_2$ *and* $\mathbf{t}_1 \longrightarrow^* \mathbf{t}'_1$*, then* $\mathbf{t}_2 \longrightarrow^* \mathbf{t}'_2$ *for some* $\mathbf{t}'_2$ *where* $\Gamma_1|\Gamma_2 \vdash \mathbf{t}'_1 \sqsubseteq_{\longrightarrow} \mathbf{t}'_2$ *(For DGG, to preserve ElabCst when reducing precision, and to preserve typing under contextual reduction of* $\mathbf{refl}(\mathbf{t}_\mathbf{w})_{\vdash \mathbf{t}_1 \cong \mathbf{t}_2}$ *)*

Lemma 5.14 (Consistency Monotone for Precision). *If* $\Gamma|\Gamma \vdash \mathbf{t}_1 \sqsubseteq_{\longrightarrow} \mathbf{t}_1'$ *and* $\Gamma|\Gamma \vdash \mathbf{t}_2 \sqsubseteq_{\longrightarrow} \mathbf{t}_2'$, *and* $\mathbf{t}_1 \cong_{\longrightarrow} \mathbf{t}_2$, *then* $\mathbf{t}_1' \cong_{\longrightarrow} \mathbf{t}_2'$ *(So reducing precision of* $\mathbf{V}$ *and* $\mathbf{V}'$ *preserves ElabCst)*

Lemma 5.15 (Structural Precision). $\sqsubseteq_{\longrightarrow}$ *contains all structural rules (For homomorphic elaboration rules to produce* $\sqsubseteq_{\longrightarrow}$*-related elaboration for* $\sqsubseteq_{\mathsf{Surf}}$*-related inputs)*

5.3 Metatheory: Proving Safety and the Gradual Guarantees

Finally, we summarize the properties that we can prove by assuming GEq satisfies the criteria of §5.2. The general idea is that each case in the proofs either (1) is the same as the proof for GCIC [Lennon-Bertrand et al. 2022] or (2) follows directly from one of our criteria. Full proofs can be found in the appendix of the extended technical report [Eremondi et al. 2022].

5.3.1 Type Safety. Type safety is shown in the usual way for operational semantics, via progress and preservation [Wright and Felleisen 1994]. Each well-typed CastEq term is either a value, or can step to a well-typed term. Confluence is necessary to prove preservation for dependent types. Space restrictions mean that the formalization of values $\mathbf{v}$ in GEq are in the appendix of the extended technical report [Eremondi et al. 2022], but the idea is to follow Lennon-Bertrand et al. [2022], adding $\mathbf{t}_1 \ \&_{\mathbf{T}} \ \mathbf{t}_2$ as value when $\mathbf{t}_1$ and $\mathbf{t}_2$ are values, neither of $\mathbf{t}_1$ and $\mathbf{t}_2$ is $?$ or $\mathfrak{U}$, and $\mathbf{T}$ is not a function type.

Proposition 5.16 (Confluence, Progress, Preservation and Elaboration).

- $\longrightarrow$ *is confluent.*
- *If* $\Gamma \vdash \mathbf{t} \Leftarrow \mathbf{T}$, *then* $\mathbf{t}$ *is a value or* $\mathbf{t} \longrightarrow \mathbf{t}'$ *for some* $\mathbf{t}'$.
- *If* $\Gamma \vdash \mathbf{t}_1 \Leftarrow \mathbf{T}$ *and* $\mathbf{t}_1 \longrightarrow \mathbf{t}_2$ *then* $\Gamma \vdash \mathbf{t}_2 \Leftarrow \mathbf{T}$.
- *If* $\Gamma \vdash t \rightharpoonup \mathbf{t} \Leftarrow \mathbf{T}$, *then* $\Gamma \vdash \mathbf{t} \Leftarrow \mathbf{T}$.

These together yield the main safety theorem.

Theorem 5.17 (Type Safety). *If* $\cdot \vdash t : T$, *then* t *has an elaboration that either steps to a normal form or steps indefinitely.*

As a corollary, we can perform inversion on the typing derivations to obtain weak canonicity. That is, every well-typed closed term that terminates steps to a canonical term of its type.

Corollary 5.18 (Weak Canonicity). *Suppose* $\cdot \vdash \mathbf{t} : \mathbf{V}$. *Then either* $\mathbf{t}$ *diverges, or* $\mathbf{t} \longrightarrow^* \mathbf{v}$ *where* $\mathbf{v}$ *is* $?_{\mathbf{V}}$ *or* $\mathfrak{U}_{\mathbf{V}}$, *or the following hold:*

- *If* $\mathbf{V}$ *is* $(\mathbf{x} : \mathbf{T}_1) \to \mathbf{T}_2$ *then* $\mathbf{v}$ *is* $\lambda \mathbf{x}.\, \mathbf{t}'$
- *If* $\mathbf{V}$ *is* $\mathbf{C}_{@\{i\}}(\overline{\mathbf{t}_1})$ *then* $\mathbf{v}$ *is* $\mathbf{D}^{\mathbf{C}}{}_{@\{i\}}(\overline{\mathbf{t}_2})$ *for some* $\mathbf{D}$.
- *If* $\mathbf{V}$ *is* $\mathbf{t}_1 =_{\mathbf{T}'} \mathbf{t}_2$ *then* $\mathbf{v}$ *is* $\mathbf{refl}(\mathbf{t}')_{\vdash \mathbf{t}_1 \cong \mathbf{t}_2}$
- *If* $\mathbf{V}$ *is* $\mathbf{Type}_{\mathbf{i}}$ *then* $\mathbf{v}$ *is one of* $\mathbf{C}_{@\{i\}}(\overline{\mathbf{t}_1})$, $(\mathbf{x} : \mathbf{T}_1) \to \mathbf{T}_2$, $\mathbf{Type}_{\mathbf{i}-1}$ *or* $\mathbf{t}_1 =_{\mathbf{T}'} \mathbf{t}_2$.

5.3.2 Conservatively Extending CIC. Each CIC rule has a direct analogue in CastEq, so it is clear that it extends CIC. Since most of the gradual-specific rules refer to $?$ or $\mathfrak{U}$, knowing that consistency collapses to α-equivalence on static terms is enough to show that said extension is conservative.

Theorem 5.19 (Conservativity). *For any BCIC-terms* t *and* T, *let* t *and* T *be the* GEq *terms corresponding to* t *and* T *by mapping BCIC* λ *to* GEq λ, *etc. Then* $\cdot \vdash \mathsf{t} \Leftarrow \mathsf{T}$ *iff* $\cdot \vdash t : T$.

5.3.3 Gradual Guarantees. To prove the gradual guarantees, we use the gradual criteria to show that elaboration is monotone. This, when combined with the monotonicity of $\rightsquigarrow$ with respect to semantic precision, gives us both the static and dynamic guarantees.

Proposition 5.20 (Elaboration Gradual Guarantee). *Suppose* $t_1 \sqsubseteq_{\text{Surf}} t_2$ *and* $\Gamma_1 \sqsubseteq_{\longrightarrow} \Gamma_2$ *(i.e. entries in* Γ_1 *and* Γ_2 *are respectively related by* $\sqsubseteq_{\longrightarrow}$*). Then:*

- *If* $\Gamma_1 \vdash t_1 \rightarrowtriangle \mathbf{t}_1 \Leftarrow \mathbf{T}$ *then* $\Gamma_2 \vdash t_2 \rightarrowtriangle \mathbf{t}_2 \Leftarrow \mathbf{T}$ *for some* $\mathbf{t}_2$ *where* $\Gamma_1 | \Gamma_2 \vdash \mathbf{t}_1 \sqsubseteq_{\longrightarrow} \mathbf{t}_2$.
- *If* $\Gamma_1 \vdash t_1 \rightarrowtriangle \mathbf{t}_1 \Rightarrow \mathbf{T}_1$ *then* $\Gamma_2 \vdash t_2 \rightarrowtriangle \mathbf{t}_2 \Rightarrow \mathbf{T}_2$ *for some* $\mathbf{T}_2, \mathbf{t}_2$ *where* $\Gamma_2 \vdash \mathbf{T}_2 \Rightarrow_{\mathbf{Type}} \mathbf{Type}_\ell$ *and* $\Gamma_1 | \Gamma_2 \vdash \mathbf{t}_1 \sqsubseteq_{\longrightarrow} \mathbf{t}_2$.

This, combined with the preservation of precision under evaluation, is enough to prove the static and dynamic gradual guarantees as stated in §5.1. The hard work lies in proving that reduction preserves precision, which we leave to §6.3.

6 CONSISTENCY AND PRECISION

Motivated by the criteria of §5.2, in this section we extend GCIC's precision and consistency relations to accommodate propositional equality and composition. We show that our relations fulfill the laws of §5.2, thus showing that GEq fulfills type safety and the gradual guarantees, justifying the design of precision and consistency.

6.1 Review: Precision and Consistency in GCIC

6.1.1 Structural Precision. Figure 10 recalls *structural precision* from GCIC [Lennon-Bertrand et al. 2022], written as $\Gamma \vdash \mathbf{t}_1 \sqsubseteq_\alpha \mathbf{t}_2$. Structural precision is the syntactic relation out of which definitional precision $\sqsubseteq_{\longrightarrow}$ is built. The *generating rules* GenUnk and GenErr establish $?_{\mathbf{T}}$ and $\mathcal{U}_{\mathbf{T}}$ as the least and most precise terms of type $\mathbf{T}$. For technical reasons, GenUnkUniv allows some cumulativity for $?_{\mathbf{Type}_j}$, while GenErrLam encodes a version of η-expansion for errors. The *diagonal rules* (named Diag*) are structural: terms are precision related if they are built with the same syntactic construct and the corresponding sub-terms are precision-related. We show a few examples, but omit most diagonal rules for space reasons. Finally, *cast rules* capture non-structural properties of casts. Rule Cast-L states that a casting $\mathbf{t}$ is more precise that $\mathbf{t}'$ if the cast's source and destination types are both more precise than the type of $\mathbf{t}'$, and if $\mathbf{t}$ is more precise that $\mathbf{t}'$. The rule Cast-R says the opposite: casting $\mathbf{t}$ is less precise than $\mathbf{t}$ if the source and destination are both less precise than the type of $\mathbf{t}$ and $\mathbf{t}$ itself is less precise than $\mathbf{t}'$.

Structural precision uses an auxiliary type judgment: *presynthesis* $\Gamma \vdash \mathbf{t} \Rightarrow^* \mathbf{T}$ is defined to be exactly the type synthesis relation without the $\sqsubseteq_{\longleftrightarrow}$ side-condition in CastRefl. Presynthesis types strictly more terms than synthesis, and both produce the same type, since they differ only in side-conditions. The side-condition is not used in the type-safety proof, so any run-time terms that presynthesize a type are safe. Unlike GCIC, GEq uses precision to type equality witnesses, so presynthesis avoids a circular dependency between typing and precision.

Structural precision is defined mutually with definitional precision (Fig. 10) $\sqsubseteq_{\longrightarrow}$, which acts as $\sqsubseteq_{\longrightarrow}$ from §5. Definitional precision allows reducing before comparing, and is used with type ascriptions, such as for functions, equality proofs and casts. Since the checking rule for CastCIC allowed arbitrary reductions, a term may be well-typed even if its type ascriptions are not fully reduced. Type ascriptions on a term may need to be reduced before structural precision is apparent. This definition is due to Lennon-Bertrand et al. [2022].

6.1.2 Syntactic Consistency. Figure 10 defines consistency for GCIC. All terms are consistent with $?$ (CstUnkL, CstUnkR). Each syntactic construct also has an (omitted) structural rule. Unlike precision, consistency between terms ignores type ascriptions, and casts are also ignored (CstCastL, CstCastR). We follow GCIC and let $\mathcal{U}_{\mathbf{T}}$ be consistent with $?_{\mathbf{T}}$.

$\boxed{\Gamma \vdash t_1 \sqsubseteq_\alpha t_2 \textit{ (Precision: Key Rules from GCIC)}}$

$$\textsc{GenUnk}\ \frac{\Gamma_1 \vdash t \Rightarrow^* T' \quad \Gamma_1|\Gamma_2 \vdash T' \sqsubseteq_{\longrightarrow} T}{\Gamma_1|\Gamma_2 \vdash t \sqsubseteq_\alpha ?_T} \qquad \textsc{GenErr}\ \frac{\Gamma_1 \vdash t \Rightarrow^* T' \quad \Gamma_1|\Gamma_2 \vdash T \sqsubseteq_{\longrightarrow} T'}{\Gamma_1|\Gamma_2 \vdash \mho_T \sqsubseteq_\alpha t}$$

$$\textsc{GenUnkUniv}\ \frac{\Gamma_1 \vdash T \Rightarrow^*_{\mathrm{Type}} \mathrm{Type}_i \quad i < j}{\Gamma_1|\Gamma_2 \vdash T \sqsubseteq_\alpha ?_{\mathrm{Type}_j}} \qquad \textsc{DiagVar}\ \frac{}{\Gamma_1|\Gamma_2 \vdash x \sqsubseteq_\alpha x}$$

$$\textsc{GenErrLam}\ \frac{\Gamma_2 \vdash t_2 \Rightarrow^*_\Pi (x : T_2) \to T_2' \quad \Gamma_1|\Gamma_2 \vdash (x : T_1) \to T_1' \sqsubseteq_{\longrightarrow} (x : T_2) \to T_2'}{\Gamma_1|\Gamma_2 \vdash \lambda(x : T_1).\, \mho_{T_1'} \sqsubseteq_\alpha t_2}$$

$$\textsc{DiagAbs}\ \frac{\Gamma_1|\Gamma_2 \vdash T_1 \sqsubseteq_{\longrightarrow} T_2 \quad \Gamma_1, (x : T_1)|\Gamma_2, (x : T_2) \vdash t_1 \sqsubseteq_\alpha t_2}{\Gamma_1|\Gamma_2 \vdash \lambda(x : T_1).\, t_1 \sqsubseteq_\alpha \lambda(x : T_2).\, t_2}$$

$$\textsc{DiagCast}\ \frac{\Gamma_1|\Gamma_2 \vdash T_1 \sqsubseteq_\alpha T_2 \qquad \Gamma_1|\Gamma_2 \vdash T_1' \sqsubseteq_\alpha T_2' \qquad \Gamma_1|\Gamma_2 \vdash t_1 \sqsubseteq_\alpha t_2}{\Gamma_1|\Gamma_2 \vdash \langle T_1' \Leftarrow T_1 \rangle t_1 \sqsubseteq_\alpha \langle T_2' \Leftarrow T_2 \rangle t_2}$$

$$\textsc{CastL}\ \frac{\Gamma_2 \vdash t_2 \Rightarrow^* T_2 \quad \Gamma_1|\Gamma_2 \vdash T_1 \sqsubseteq_{\longrightarrow} T_2 \quad \Gamma_1|\Gamma_2 \vdash T_1' \sqsubseteq_{\longrightarrow} T_2 \quad \Gamma_1|\Gamma_2 \vdash t_1 \sqsubseteq_\alpha t_2}{\Gamma_1|\Gamma_2 \vdash \langle T_1' \Leftarrow T_1 \rangle t_1 \sqsubseteq_\alpha t_2}$$

$$\textsc{CastR}\ \frac{\Gamma_1 \vdash t_1 \Rightarrow^* T_1 \quad \Gamma_1|\Gamma_2 \vdash T_1 \sqsubseteq_{\longrightarrow} T_2 \quad \Gamma_1|\Gamma_2 \vdash T_1 \sqsubseteq_{\longrightarrow} T_2' \quad \Gamma_1|\Gamma_2 \vdash t_1 \sqsubseteq_\alpha t_2}{\Gamma_1|\Gamma_2 \vdash t_1 \sqsubseteq_\alpha \langle T_2' \Leftarrow T_2 \rangle t_2}$$

$\boxed{\Gamma_1|\Gamma_2 \vdash t_1 \sqsubseteq_{\longrightarrow} t_2 \textit{ (Definitional Precision)}}$

$$\textsc{DefBase}\ \frac{\Gamma_1|\Gamma_2 \vdash t_1 \sqsubseteq_\alpha t_2}{\Gamma_1|\Gamma_2 \vdash t_1 \sqsubseteq_{\longrightarrow} t_2} \qquad \textsc{DefStepL}\ \frac{t_1 \longrightarrow t_1' \quad \Gamma_1|\Gamma_2 \vdash t_1' \sqsubseteq_{\longrightarrow} t_2}{\Gamma_1|\Gamma_2 \vdash t_1 \sqsubseteq_{\longrightarrow} t_2}$$

$$\textsc{DefStepR}\ \frac{t_2 \longrightarrow t_2' \quad \Gamma_1|\Gamma_2 \vdash t_1 \sqsubseteq_{\longrightarrow} t_2'}{\Gamma_1|\Gamma_2 \vdash t_1 \sqsubseteq_{\longrightarrow} t_2}$$

$\boxed{t_1 \cong_\alpha t_2 \textit{ (Consistency: Non-structural Rules)}}$

$$\textsc{CstVar}\ \frac{}{x \cong_\alpha x} \qquad \textsc{CstUnkL}\ \frac{}{?_T \cong_\alpha T} \qquad \textsc{CstUnkR}\ \frac{}{t \cong_\alpha ?_T}$$

$$\textsc{CstCastL}\ \frac{t \cong_\alpha t'}{\langle T_2 \Leftarrow T_1 \rangle t \cong_\alpha t'} \qquad \textsc{CstCastR}\ \frac{t \cong_\alpha t'}{t \cong_\alpha \langle T_2 \Leftarrow T_1 \rangle t'}$$

Fig. 10. Structural Precision and Consistency for GCIC: Key Rules

6.2 Precision and Consistency for GEq

The structural precision laws are not sufficient for handling composition. In particular, we want $\Gamma|\Gamma' \vdash t_1 \mathbin{\&_T} t_2 \sqsubseteq t_1$, with the same holding for t_2. However, this fact is not derivable from the diagonal rule for composition. Instead, we must add rules to ensure that composing produces a lower bound. However, once we start adding non-structural rules, we must be careful not to disrupt the other properties we need from precision. For example, §5.2 states that precision must be transitive. If we have $(x \mathbin{\&_T} y) \mathbin{\&_T} z \sqsubseteq x \mathbin{\&_T} y$ and $x \mathbin{\&_T} y \sqsubseteq x$, but we also want $(x \mathbin{\&_T} y) \mathbin{\&_T} z \sqsubseteq x$, then we must transitively apply the fact that composing produces a lower bound.

Figure 11 shows the added rules. DiagRefl and DiagComp, along with the omitted DiagEq and DiagJ, are like the other diagonal rules. The rules PrecCompL and PrecCompR encode that the composition is a precision-lower bound, but in a way that preserves transitivity. The rules for $\sqsubseteq_{\longrightarrow}^{\longleftarrow}$ are also shown: like $\sqsubseteq_{\longrightarrow}$ they allow for reductions before comparing with structural precision, but they also allow backwards steps, fulfilling Lemma 5.9. We only allow backwards-steps for the less precise term, since backwards steps on the left-hand are admissible by Lemma 5.13.

$$\textsc{DiagRefl}\ \frac{\Gamma_1|\Gamma_2 \vdash \mathbf{t_1} \sqsubseteq_\alpha \mathbf{t_2}}{\Gamma_1|\Gamma_2 \vdash \mathbf{t_1'} \sqsubseteq_{\longrightarrow} \mathbf{t_2'} \quad \Gamma_1|\Gamma_2 \vdash \mathbf{t_1''} \sqsubseteq_{\longrightarrow} \mathbf{t_2''}}{\Gamma_1|\Gamma_2 \vdash \mathbf{refl(t_1)}_{\vdash \mathbf{t_1'} \cong \mathbf{t_1''}} \sqsubseteq_\alpha \mathbf{refl(t_2)}_{\vdash \mathbf{t_2'} \cong \mathbf{t_2''}}}$$

$$\textsc{DiagComp}\ \frac{\Gamma_1|\Gamma_2 \vdash \mathbf{T_1} \sqsubseteq_\alpha \mathbf{T_2} \quad \Gamma_1|\Gamma_2 \vdash \mathbf{t_1} \sqsubseteq_\alpha \mathbf{t_2} \quad \Gamma_1|\Gamma_2 \vdash \mathbf{t_1'} \sqsubseteq_\alpha \mathbf{t_2'}}{\Gamma_1|\Gamma_2 \vdash \mathbf{t_1} \mathbin{\&_{T_1}} \mathbf{t_1'} \sqsubseteq_\alpha \mathbf{t_2} \mathbin{\&_{T_2}} \mathbf{t_2'}}$$

$$\textsc{PrecCompL}\ \frac{\Gamma_1|\Gamma_2 \vdash \mathbf{t_1} \sqsubseteq_\alpha \mathbf{t_3}}{\Gamma_1|\Gamma_2 \vdash \mathbf{t_1} \mathbin{\&_T} \mathbf{t_2} \sqsubseteq_\alpha \mathbf{t_3}} \qquad \textsc{PrecCompR}\ \frac{\Gamma_1|\Gamma_2 \vdash \mathbf{t_2} \sqsubseteq_\alpha \mathbf{t_3}}{\Gamma_1|\Gamma_2 \vdash \mathbf{t_1} \mathbin{\&_T} \mathbf{t_2} \sqsubseteq_\alpha \mathbf{t_3}}$$

$$\textsc{CstComp(L,R)}\ \frac{\mathbf{t_1} \cong_\alpha \mathbf{t_3} \quad \mathbf{t_2} \cong_\alpha \mathbf{t_3}}{\mathbf{t_1} \mathbin{\&_T} \mathbf{t_2} \cong_\alpha \mathbf{t_3} \quad \mathbf{t_3} \cong_\alpha \mathbf{t_1} \mathbin{\&_T} \mathbf{t_2}} \qquad \textsc{CstCompDiag}\ \frac{\mathbf{t_1} \cong_\alpha \mathbf{t_1'} \quad \mathbf{t_2} \cong_\alpha \mathbf{t_2'}}{\mathbf{t_1} \mathbin{\&_T} \mathbf{t_2} \cong_\alpha \mathbf{t_1'} \mathbin{\&_{T'}} \mathbf{t_2'}}$$

$$\textsc{PrecModStruct}\ \frac{\Gamma_1|\Gamma_2 \vdash \mathbf{t_1} \sqsubseteq_\alpha \mathbf{t_2}}{\Gamma_1|\Gamma_2 \vdash \mathbf{t_1} \sqsubseteq_{\longrightarrow}^{\longleftarrow} \mathbf{t_2}} \qquad \textsc{PrecModStepL}\ \frac{\mathbf{t_1} \longrightarrow^* \mathbf{t_1'} \quad \Gamma_1|\Gamma_2 \vdash \mathbf{t_1'} \sqsubseteq_{\longrightarrow}^{\longleftarrow} \mathbf{t_2}}{\Gamma_1|\Gamma_2 \vdash \mathbf{t_1} \sqsubseteq_{\longrightarrow}^{\longleftarrow} \mathbf{t_2}}$$

$$\textsc{PrecModStepR}\ \frac{\mathbf{t_2} \longrightarrow^* \mathbf{t_2'} \quad \Gamma_1|\Gamma_2 \vdash \mathbf{t_1} \sqsubseteq_{\longrightarrow}^{\longleftarrow} \mathbf{t_2'}}{\Gamma_1|\Gamma_2 \vdash \mathbf{t_1} \sqsubseteq_{\longrightarrow}^{\longleftarrow} \mathbf{t_2}} \qquad \textsc{PrecModStepBack}\ \frac{\mathbf{t_2'} \longrightarrow^* \mathbf{t_2} \quad \Gamma_1|\Gamma_2 \vdash \mathbf{t_1} \sqsubseteq_{\longrightarrow}^{\longleftarrow} \mathbf{t_2'}}{\Gamma_1|\Gamma_2 \vdash \mathbf{t_1} \sqsubseteq_{\longrightarrow}^{\longleftarrow} \mathbf{t_2}}$$

Fig. 11. CastEq Precision and Consistency rules for composition

Last are the rules for (static) consistency for composition. Recall that §5.2 requires that reducing precision preserves consistency. Since composition is as precise as both its arguments, $\mathbf{t_1} \mathbin{\&_T} \mathbf{t_2} \cong_\alpha \mathbf{t_3}$ should imply that $\mathbf{t_1}$ and $\mathbf{t_2}$ are both consistent with $\mathbf{t_3}$. We conjecture that composition is a (semantic) greatest lower bound, which would mean that errors in composing witnesses are never flagged earlier than necessary. For this to hold, the composition of two terms must be consistent with everything that is consistent with both of those two terms. Our composition consistency rules in Fig. 11 express this: CstCompL and CstCompR ensure that $\mathbf{t_1} \mathbin{\&_T} \mathbf{t_2}$ is consistent with exactly the terms that are consistent with both $\mathbf{t_1}$ and $\mathbf{t_2}$.

With consistency fully defined, the difference between static and dynamic consistency is now clear: two terms that share a non-error lower-bound may be statically inconsistent if they differ only in neutral terms. Variables are only statically consistent with themselves (CstVar) or $?$ (CstUnkL,R). However, for any two variables $\mathbf{x}$ and $\mathbf{y}$, $\mathbf{x} \mathbin{\&_T} \mathbf{y}$ is a non-error term that is as precise as both, as given by PrecComp(L,R). This disconnect between precision and consistency is justified by the criteria of §5.2: we show below that reducing precision preserves consistency. The separation of static and dynamic consistency enables the gradual guarantees and conservatively embedding CIC while maintaining static equivalences.

6.3 Fulfilling The Critera

GEq is now fully defined: we have defined the precision relations $\sqsubseteq_{\longrightarrow}$ and $\sqsubseteq_{\longrightarrow}^{\longleftarrow}$ and the consistency relation $\cong_{\longrightarrow}$ to instantiate $\boxed{\sqsubseteq_{\longrightarrow}}$, $\boxed{\sqsubseteq_{\longrightarrow}^{\longleftarrow}}$ and $\boxed{\cong_{\longrightarrow}}$. We now establish that these relations fulfill the criteria of §5.2. We give the intuition behind some of the cases that are new compared to GCIC. Full proofs can be found in the appendix of the extended technical report [Eremondi et al. 2022].

- **Immediate Results:** Proving reflexivity of $\sqsubseteq_{\longrightarrow}$ (Lemma 5.4) is a straightforward induction. The rules PrecCompL and PrecCompR make the composition of terms as precise as either term, proving Lemma 5.7. The closure of $\sqsubseteq_{\longrightarrow}^{\longleftarrow}$ under convertibility is built into its definition, proving Lemma 5.9. DiagCast gives that casts are monotone, proving Lemma 5.11. The monotonicity of substitution (Lemma 5.12) is proved with a straightforward induction, relying on presynthesis preserving types under substitution. The remaining diagonal rules give that $\sqsubseteq_{\longrightarrow}$ has all structural rules, fulfilling Lemma 5.15.
- **Composition Safety** (Lemma 5.5) For progress, each composition of two canonical forms of the same type has a reduction, where either the heads match and the arguments are composed, or an

error is produced. If one of the composed terms is not a canonical form, then either (1) one of the composed terms can reduce, (2) one term is a $?_{\mathbf{T}}$ or $\mho_{\mathbf{T}}$ where $\mathbf{T}$ is not a function or equality type, and we can reduce with RedCompUnk or RedCompErr, or (3) one of the composed terms is neutral, and hence the composition is neutral. For preservation, either the result is immediate, or casts are inserted to ensure that types are preserved.

- **Composition Confluence** (Lemma 5.6) RedCompUnk(L,R) ensures that composing with $?_{\mathbf{T}}$ only reduces when $?_{\mathbf{T}}$ cannot reduce, avoiding a "diamond" problem.
- **Precision Transitive** (Lemma 5.8): We prove this *after* monotonicity of reduction, which lets us prove that precision-related types have precision-related terms, which is necessary to fulfill premises on terms' types, such as in CastL and CastR. The rest is straightforward induction.
- **Static Consistency** (Lemma 5.10) The GEq rules not present in GCIC are for equality, which are trivially handled, and consistency rules for composition, for which the result vacuously holds since composition is not present in the static language.
- **Monotonicity of Reduction** (Lemma 5.13): The key fact is that, since PrecCompL and PrecCompR only have composition on the left, all the inversions in the GCIC proofs are still valid for GEq. The interesting case is when precision is derived using PrecCompL (PrecCompR is symmetric), and the composition reduces. The result is trivial for RedCompUnk(L,R) and RedCompErr(L,R). For the remaining cases, two terms with the same head are being composed, and the result is either $\mho$ or another term with the same head. When $\mho$ is produced the result is trivial. When a term with the same head is produced, the PrecCompL can be used with the appropriate diagonal rule. In the case that casts are present in the result of composition, CastL is used. The other notable case is when $\mathbf{J}$ reduces, where the result is derived using DiagCast.
- **Consistency Monotone** (Lemma 5.14) We first show that consistency is monotone on the left, then prove that it is symmetric to obtain monotonicity for both arguments. The case when $\cong_\alpha$ is derived with CstCompR or CstCastR must be handled specially, since they each take an operand that can be any term. The trick is to unwrap the chain of CstCompR and CstCastR uses, use the induction hypothesis on the contained derivation, then re-apply CstCompR and CstCastR in the same order to obtain the result. When precision is derived with PrecCompL or PrecCompR, then consistency must have either been derived with CstCompDiag, in which case the result follows from the induction hypothesis, or with CstCompL or (symmetrically) CstCompR. For CstCompL, the premise gives that both composed terms are consistent with the right-hand term, yielding our result. The remaining cases are straightforward.

7 DISCUSSION

7.1 Extensions Enabled by Equality

In addition to catching the kinds of bugs discussed in §2, we show some benefits of having propositional equality in GEq. Three new language features can be encoded using propositional equality, without augmenting the cast calculus: empty types, Axiom K, and indexed inductive type families. For type families, we discuss some limitations of the approach and workarounds for these limitations, showing how our cast calculus is expressive enough to pave the way for future improvements.

7.1.1 The Empty Type. Just as the gradual $\mathbf{J}$ needed computation, eliminating the empty type has computational content in a gradual language. In static languages, the empty type Empty has no closed values, so either Empty contains no terms, or (for logically inconsistent languages) any such terms are non-terminating. The elimination function $\mathsf{exfalso} : (\mathsf{X} : \mathbf{Type}_\mathsf{i}) \to \mathsf{Empty} \to \mathsf{X}$ produces a result of any type, given a value of the empty type. In a gradual language, however, $?$ and $\mho$ can be used at any type, including the empty type. So a gradual **exfalso** must produce a value of type $\mathbf{X}$.

We again follow the goal of dynamically tracking constraints expressed by types. For the empty type, a value of type $\mathbf{f} : \mathbf{T} \to \mathbf{Empty}$ encodes the constraint that $\mathbf{T}$ should be impossible, and a branch built using **exfalso** should be unreachable. If $\mathbf{f}$ is applied to $\mathbf{t} : \mathbf{Empty}$, created using $?$ or casts, then the constraint has been violated, and an error should be raised.

We can encode this behavior by defining **Empty** to be $\mathbf{true} =_{\mathbb{B}} \mathbf{false}$, and **exfalso** to be $\lambda \mathbf{X}.\, \lambda \mathbf{t}.\, \mathbf{J}\ (\mathbf{b}.\, \mathbf{if}\ \mathbf{b}\ \mathbb{B}\ \mathbf{X})\ \mathbf{true}\ \mathbf{t}$. The key is $?_{\mathbf{Empty}}$ and $\mathcal{U}_{\mathbf{Empty}}$ both evaluate to $\mathbf{refl}(\mathcal{U}_{\mathbb{B}})_{\vdash \mathbf{true} \cong \mathbf{false}}$. So the only value of type **Empty** is a dynamic type error. Likewise, the eliminator **exfalso** casts $\mathbf{t}$ to type $\mathcal{U}_{\mathbf{Type}_\ell}$ before casting it to type $\mathbf{X}$, so the result is always $\mathcal{U}_{\mathbf{X}}$. Without adding any features to CastEq, the bug-finding described in §2 handles constraints encoded as logical negation.

7.1.2 Axiom K. Because $?_{\mathbf{t}_1 =_{\mathbf{T}} \mathbf{t}_2}$ steps to $\mathbf{refl}(\mathbf{t}_1 \ \&_{\mathbf{T}}\ \mathbf{g}_2)_{\vdash \mathbf{t}_1 \cong \mathbf{t}_2}$, GEq is in the class of dependently typed languages where **refl** is the only constructor for equality. Composition can be used to derive a (gradual) proof of this uniqueness, even though no such proof can be derived in most static type theories [Hofmann and Streicher 1998]:

$$\mathbf{K} : (\mathbf{x} : \mathbf{T}) \to (\mathbf{pf} : \mathbf{x} =_{\mathbf{T}} \mathbf{x}) \to \mathbf{pf} =_{\mathbf{x} =_{\mathbf{T}} \mathbf{x}} \mathbf{refl}(\mathbf{x})_{\vdash \mathbf{x} \cong \mathbf{x}}$$
$$\mathbf{K}\ \mathbf{x}\ \mathbf{pf} = \mathbf{refl}(\mathbf{pf}\ \&_{\mathbf{x} =_{\mathbf{T}} \mathbf{x}}\ \mathbf{refl}(\mathbf{x})_{\vdash \mathbf{x} \cong \mathbf{x}})_{\vdash \mathbf{pf} \cong \mathbf{refl}(\mathbf{x})_{\vdash \mathbf{x} \cong \mathbf{x}}}$$

Axiom K can be used to prove that all equality proofs of a given type are equal [Streicher 1993], so our proof-irrelevant **J** principle does not lose any expressivity, since any types parameterized by an equality proof can be rewritten with **K**. Also, Axiom K allows for conventional dependent pattern matching to be elaborated into inductive eliminators [Goguen et al. 2006], providing a lightweight alternative to the cumbersome `ind` form. The combination of Axiom K and function extensionality suggests a connection to Observational Type Theory [Altenkirch et al. 2007; Pujet and Tabareau 2022] that warrants future exploration.

7.1.3 Inductive Types. McBride [2000] describes how, using propositional equality, indexed inductive families can be encoded. The main idea is, instead of having each constructor return different indices, each index is a parameter, and each constructor takes an equality proof that the parameter has the desired value. In the elimination principle, **J** is used to rewrite the type of the returned value using the stored equality. Consider how the classic vector type is transformed:

$$\begin{array}{l}\mathbf{data}\ \mathbf{Vec}\ (\mathbf{X} : \mathbf{Type}) : (\mathbf{n} : \mathbb{N}) \to \mathbf{Type}\ \mathit{where} \\ \quad \mathbf{Nil} : \mathbf{Vec}\ \mathbf{X}\ 0 \\ \quad \mathbf{Cons} : \mathbf{X} \to \mathbf{Vec}\ \mathbf{X}\ \mathbf{n} \to \mathbf{Vec}\ \mathbf{X}\ (1 + \mathbf{n})\end{array}$$

$$\begin{array}{l}\mathbf{data}\ \mathbf{Vec'}(\mathbf{X} : \mathbf{Type})(\mathbf{n} : \mathbb{N}) : \mathbf{Type}\ \mathit{where} \\ \quad \mathbf{Nil'} : (\mathbf{n} =_{\mathbb{N}} 0) \to \mathbf{Vec}\ \mathbf{X}\ \mathbf{y} \\ \quad \mathbf{Cons'} : (\mathbf{z} : \mathbb{N}) \to \mathbf{X} \to \mathbf{Vec}\ \mathbf{X}\ \mathbf{z} \\ \qquad \to \mathbf{n} =_{\mathbb{N}} (1 + \mathbf{z}) \to \mathbf{Vec}\ \mathbf{X}\ \mathbf{n}\end{array}$$

This transformation gives a low-overhead way to incorporate indexed inductive families with gradual dependent types. Since no extensions to CastEq are required, the safety and gradual guarantee results from §5 apply. The constructors take equality proofs, so violations of those equalities raise dynamic type errors.

However, the approach is limited in its ability to eagerly detect errors. The problem is that dynamic consistency is fundamentally not transitive, since otherwise all types are consistent through $?$. Members of inductive types are essentially trees, and equality constraints track constraints at each level of the tree, but consistency across the entire tree is not ensured. The witnesses track the evolution of type information across time, but not across space. Consider $\mathbf{Cons'}\ ?_{\mathbb{N}}\ \mathbf{true}\ (\mathbf{Nil'}\ ?_{\mathbb{N}}\ \mathbf{refl}(0)_{\vdash ?_{\mathbb{N}} 0 \cong})\ \mathbf{refl}(2)_{\vdash 2 \cong ?_{\mathbb{N}}} : \mathbf{Vec'}\ \mathbb{B}\ 2$, a vector with one element, whose type says it has length 2. Constructing this vector raises no run-time type errors. At each level, the equality proof is correct: 0 is consistent with $?_{\mathbb{N}}$, and 2 is consistent with $1 + ?_{\mathbb{N}}$. Gradually, the non-transitivity means that imprecision at each level can cause disconnects between levels.

Thankfully, CastEq is expressive enough to encode a solution to this problem. By having composition as an operator in the language, one can define so-called "smart constructors" that have the same types as the normal constructors, but that access the equality proofs stored in the previous level of the tree when constructing new ones. For example, using $\mathbf{J}$ we can write $\mathbf{cong1} : (\mathbf{m}\ \mathbf{n} : \mathbb{N}) \to \mathbf{m} =_{\mathbb{N}} \mathbf{n} \to 1 + \mathbf{m} = 1 + \mathbf{n}$, which can be used in a "smart" $\mathbf{Cons}$:

$$\mathbf{smartCons\ z\ h\ t\ eq} = \mathbf{Cons\ z\ h\ t\ (eq\ \&_{(1+z)=_{\mathbb{N}}n}\ (\langle (1+z) =_{\mathbb{N}} n \Leftarrow 1+z =_{\mathbb{N}} 1+z \rangle (cong1\ (wit\ t))))}$$

where $\mathbf{wit} : \mathbf{Vec\ X\ z} \to \mathbf{z} =_{\mathbb{N}} \mathbf{z}$

$$\mathbf{wit\ (Nil\ eq)} = \langle \mathbf{z} =_{\mathbb{N}} \mathbf{z} \Leftarrow 0 =_{\mathbb{N}} \mathbf{z} \rangle \mathbf{eq} \quad | \quad \mathbf{wit\ (Cons\ x\ h\ t\ eq)} = \langle \mathbf{z} =_{\mathbb{N}} \mathbf{z} \Leftarrow 1+\mathbf{x} =_{\mathbb{N}} \mathbf{z} \rangle \mathbf{eq}$$

When $\mathbf{smartCons}$ is used in place of $\mathbf{Cons'}$, the witness $\mathbf{refl}(0)_{\vdash ?_{\mathbb{N}} \cong 0}$ is transformed to $\mathbf{refl}(1)_{\vdash ?_{\mathbb{N}} \cong ?_{\mathbb{N}}}$, which produces an error when cast to $1+?_{\mathbb{N}} =_{\mathbb{N}} 2$, since $1\ \&_{\mathbb{N}}\ 2 \rightsquigarrow \mathbb{U}_{\mathbb{N}}$. Formalizing the general version of this approach is beyond the scope of this paper, but it shows how having composition as an operator enables more detailed manipulation of run-time type information.

7.2 Future Work

7.2.1 Termination and Approximate Normalization. As presented, GEq has undecidable type checking. Precision and CastEq typing are not obviously decidable. Thankfully, we do not need to decide precision typing to decide GEq typing, since initial witnesses are always valid by reflexivity. Deciding GEq typing is straightforward, except for normalization: some terms do not terminate and consistency compares modulo reduction. In GCIC, Lennon-Bertrand et al. [2022] show that termination can be obtained by sacrificing the gradual guarantees, or by restricting universes so that they are not closed under function types. While useful for type theory, these sacrifices remove reasoning principles or reduce expressivity, respectively, that make them unsuitable for programming.

Eremondi et al. [2019] propose *approximate normalization*, with different semantics for compile-time normalization of types and run-time evaluation of terms. At compile-time, when missing type information means that termination cannot be guaranteed, $?$ is produced as an approximation. Run-time evaluation uses no approximations, so expressivity is not lost. We conjecture that approximate normalization can be easily added to GEq. The challenge is finding a suitable termination argument, since Eremondi et al. [2019] provide a proof that does not apply to inductive types. The syntactic-model strategy of Lennon-Bertrand et al. [2022] can likely be adapted. Also, a decision must be made about whether approximate or exact normalization should be used for run-time witness calculations: since they are opaque to the programmer, witnesses may obscure the causes of non-termination.

7.2.2 Conjectures: EP Pairs, Composition, Blame and Full Abstraction. Lennon-Bertrand et al. [2022] prove a stronger property than the gradual guarantees for GCIC. They show that casts between precision-related types form an embedding-projection (EP) pair [New and Ahmed 2018], so that increasing then decreasing precision produces the same result modulo errors, and decreasing then increasing precision produces an observationally-equivalent result. While the gradual guarantees are helpful, they are satisfied by trivial languages where every cast produces $?$. Showing the EP pair property would prove that casts in GEq never lose run-time information, giving more confidence in its ability to dynamically track constraints. We conjecture that GEq fulfills the EP pair property, but suspect novel proof techniques are needed to handle witness proofs. We also conjecture that composition computes the greatest lower-bound for semantic precision, so that each type forms a true semi-lattice. This would establish that witness composition never prematurely raises dynamic errors, since two witnesses would compose to $\mathbb{U}$ only when all other options are impossible.

Another desirable property of gradual languages is a blame theorem [Wadler and Findler 2009], stating that, for a dynamic cast error, the less-precise type is always blamed. GEq has no notion of blame, but we conjecture that the techniques of Zalewski et al. [2020] could be adapted to GEq.

Finally, we conjecture that there is a variant of CIC whose embedding into GEq is fully abstract, meeting the criteria Jacobs et al. [2021] set out for gradual languages. Intuitively, we can form equalities between extensionally-equal functions, and use those to cast between types indexed by them, so any property of a function should apply to one that is extensionally-equal. Full abstraction guarantees that all static equivalences hold in GEq, giving the programmer more tools with which to reason about their code. Proving full abstraction for non-dependently typed gradual languages is a recent development, so more investigation is needed to adapt these techniques to dependent types. The usual technique for full abstraction is to simulate the target language in the source, so every target context can be translated into a source context that is unable to distinguish the terms. As a consequence, the embedded variant of CIC must have capabilities for non-termination added.

7.3 Related Work

Gradual Approaches to Equality: GRIP [Maillard et al. 2022] extends CastCIC with propositional equality, but unlike GEq, this propositional equality is in a separate sort, the types of which have all members definitionally equal. This propositional layer has no imprecision, and contains features for catching dynamic type errors in the gradual layer. GRIP features an *internal* notion of precision: while not all terms obey the gradual guarantees, self-precise terms do, and guarantees for such terms are available as a theorem in the propositional layer. GEq provides reasoning about equality *within* gradual programs, whereas GRIP provides a layer for reasoning *about* gradual programs. GRIP is based on observational type theory (OTT), and hence supports fully static proofs of function extensionality and Axiom K. GEq contains only gradual proofs of these. We conjecture that, like in GRIP, OTT could replace BCIC as the underlying static language for GEq.

Our work also relates to that of Lemay [2022], which presents a dependently typed language where all definitional equality checks are deferred to run time. Like in GCIC and GEq, casts are inserted when a term's synthesized type differs from the type against which it is checked. However, there is no unknown term/type, and no requirement that synthesized and checked-against types be consistent, so errors are raised only when safety is directly violated. Like in GEq, function equality is checked extensionally: casts accumulate on unapplied functions, and arguments are cast when cast-containing functions are applied, similar to composing functions in GEq. The work emphasizes clear error messages, and unlike GEq, features a rich notion of blame.

Flexible Dependent Types: GEq builds on a long line of work mixing dynamic and static enforcement of specifications. Ou et al. [2004] support mixed static and dynamic checking of boolean-valued properties, and Lehmann and Tanter [2017] provide gradual typing for refinement types. Similarly, Tanter and Tabareau [2015] develop a system of casts for Coq, using an unsound axiom to represent type errors. These casts worked for values of subset types, i.e. a value paired with a proof that some boolean-valued function returns true for that value, but not general inductive types. Osera et al. [2012] present *dependent interoperability* for principled mixing of dependently typed and non-dependently typed programs. Dependent interoperability was extended by Dagand et al. [2016, 2018], who provide a general mechanism for lifting higher-order programs to the dependently typed setting. All of these approaches presuppose separate simple and dependent versions of types, related by boolean-valued predicates. Our composition of witnesses provides similar checks, but by keeping witnesses, types need not be reformulated in terms of subset types or boolean predicates.

Acknowledgments: We thank Felipe Bañados Schwerter, Paulette Koronkevich, Jonathan Chan, Joanne Traves, and the anonymous reviewers for their feedback on this work.

REFERENCES

Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. 2007. Observational Equality, Now!. In Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification (Freiburg, Germany) (PLPV '07). ACM, New York, NY, USA, 57–68. https://doi.org/10.1145/1292597.1292608

Felipe Bañados Schwerter, Alison M. Clark, Khurram A. Jafery, and Ronald Garcia. 2021. Abstracting Gradual Typing Moving Forward: Precise and Space-Efficient. Proc. ACM Program. Lang. 5, POPL, Article 61 (Jan. 2021), 28 pages. https://doi.org/10.1145/3434342

Edwin Brady. 2017. Type-driven development with Idris. Manning. https://www.manning.com/books/type-driven-development-with-idris

Pierre-Évariste 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. https://doi.org/10.1145/2951913.2951933

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

Joseph Eremondi, Ronald Garcia, and Éric Tanter. 2022. Propositional Equality for Gradual Dependently Typed Programming (Extended Technical Report). https://doi.org/10.48550/ARXIV.2205.01241

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

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-order Functions. In Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (Pittsburgh, PA, USA) (ICFP '02). ACM, New York, NY, USA, 48–59. https://doi.org/10.1145/581478.581484

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 (St. Petersburg, FL, USA) (POPL '16). ACM, New York, NY, USA, 429–442. https://doi.org/10.1145/2837614.2837670

Healfdene Goguen, Conor McBride, and James McKinna. 2006. Eliminating Dependent Pattern Matching. Springer Berlin Heidelberg, Berlin, Heidelberg, 521–540. https://doi.org/10.1007/11780274_27

Martin Hofmann and Thomas Streicher. 1998. The groupoid interpretation of type theory. Twenty-five years of constructive type theory (Venice, 1995) 36 (1998), 83–111. https://doi.org/10.1093/oso/9780198501275.003.0008

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

Nico Lehmann and Éric Tanter. 2017. Gradual Refinement Types. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (Paris, France) (POPL 2017). ACM, New York, NY, USA, 775–788. https://doi.org/10.1145/3009837.3009856

Mark Lemay. 2022. A Dependently Typed Programming Language With Dynamic Equality. Ph. D. Dissertation. Boston University. https://github.com/marklemay/thesis

Meven Lennon-Bertrand. 2021. Complete Bidirectional Typing for the Calculus of Inductive Constructions. In 12th International Conference on Interactive Theorem Proving (ITP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 193), Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ITP.2021.24

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

Kenji Maillard, Meven Lennon-Bertrand, Nicolas Tabareau, and Éric Tanter. 2022. A Reasonably Gradual Type Theory. Proc. ACM Program. Lang. 6, ICFP (2022). https://doi.org/10.1145/3547655 Preprint: https://hal.inria.fr/hal-03596652/.

Per Martin-Löf. 1975. About Models for Intuitionistic Type Theories and the Notion of Definitional Equality. In Proceedings of the Third Scandinavian Logic Symposium, Stig Kanger (Ed.). Studies in Logic and the Foundations of Mathematics, Vol. 82. Elsevier, 81 – 109. https://doi.org/10.1016/S0049-237X(08)70727-4

Per Martin-Löf. 1982. Constructive Mathematics and Computer Programming. In Logic, Methodology and Philosophy of Science VI, L. Jonathan Cohen, Jerzy Łoś, Helmut Pfeiffer, and Klaus-Peter Podewski (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 104. Elsevier, 153–175. https://doi.org/10.1016/S0049-237X(09)70189-2

Conor McBride. 2000. Dependently typed functional programs and their proofs. Ph. D. Dissertation. University of Edinburgh, UK. http://hdl.handle.net/1842/374

Conor McBride. 2002. Elimination with a Motive. In Types for Proofs and Programs (Berlin, Heidelberg, 2002) (Lecture Notes in Computer Science), Paul Callaghan, Zhaohui Luo, James McKinna, Robert Pollack, and Robert Pollack (Eds.). Springer, 197–216. https://doi.org/10.1007/3-540-45842-5_13

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

Peter-Michael Osera, Vilhelm Sjöberg, and Steve Zdancewic. 2012. Dependent Interoperability. In Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification (Philadelphia, Pennsylvania, USA) (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. https://doi.org/10.1007/1-4020-8141-3_34

Loïc Pujet and Nicolas Tabareau. 2022. Observational Equality: Now for Good. Proc. ACM Program. Lang. 6, POPL, Article 32 (jan 2022), 27 pages. https://doi.org/10.1145/3498693

Jeremy Siek, Peter Thiemann, and Philip Wadler. 2015a. Blame and Coercion: Together Again for the First Time. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (Portland, OR, USA) (PLDI '15). Association for Computing Machinery, New York, NY, USA, 425–435. https://doi.org/10.1145/2737924.2737968

Jeremy G. Siek and Tianyu Chen. 2021. Parameterized cast calculi and reusable meta-theory for gradually typed lambda calculi. Journal of Functional Programming 31 (2021), e30. https://doi.org/10.1017/S0956796821000241

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In Scheme and Functional Programming Workshop. 81–92. http://scheme2006.cs.uchicago.edu/scheme2006.pdf

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

Jeremy G. Siek and Philip Wadler. 2010. Threesomes, with and Without Blame. In Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Madrid, Spain) (POPL '10). ACM, New York, NY, USA, 365–376. https://doi.org/10.1145/1706299.1706342

Thomas Streicher. 1993. Investigations into intensional type theory. Ph. D. Dissertation. Ludwig Maximilian Universität. https://www2.mathematik.tu-darmstadt.de/~streicher/HabilStreicher.pdf Habilitation thesis.

M. Takahashi. 1995. Parallel Reductions in λ-Calculus. Information and Computation 118, 1 (1995), 120–127. https://doi.org/10.1006/inco.1995.1057

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

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009 (York, UK) (ESOP '09). Springer-Verlag, Berlin, Heidelberg, 1–16. https://doi.org/10.1007/978-3-642-00590-9_1

A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. Information and Computation 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093

Jakub Zalewski, James McKinna, J. Garrett Morris, and Philip Wadler. 2020. λdB: Blame tracking at higher fidelity. https://wgt20.irif.fr/wgt20-final98-acmpaginated.pdf First ACM SIGPLAN Workshop on Gradual Typing 2020, WGT 2020 ; Conference date: 19-01-2020 Through 25-01-2020.