A Reasonably Gradual Type Theory

Kenji Maillard, Meven Lennon-Bertrand, Nicolas Tabareau, Éric Tanter

A Reasonably Gradual Type Theory

KENJI MAILLARD, Gallinette Project-Team, Inria, France
MEVEN LENNON-BERTRAND, Gallinette Project-Team, Inria, France
NICOLAS TABAREAU, Gallinette Project-Team, Inria, France
ÉRIC TANTER, PLEIAD Lab, Computer Science Department (DCC), University of Chile, Chile

Gradualizing the Calculus of Inductive Constructions (CIC) involves dealing with subtle tensions between normalization, graduality, and conservativity with respect to CIC. Recently, GCIC has been proposed as a parametrized gradual type theory that admits three variants, each sacrificing one of these properties. For devising a gradual proof assistant based on CIC, normalization and conservativity with respect to CIC are key, but the tension with graduality needs to be addressed. Additionally, several challenges remain: (1) The presence of two wildcard terms at any type—the error and unknown terms—enables trivial proofs of any theorem, jeopardizing the use of a gradual type theory in a proof assistant; (2) Supporting general indexed inductive families, most prominently equality, is an open problem; (3) Theoretical accounts of gradual typing and graduality so far do not support handling type mismatches detected during reduction; (4) Precision and graduality are external notions not amenable to reasoning within a gradual type theory. All these issues manifest primally in CastCIC, the cast calculus used to define GCIC. In this work, we present an alternative to CastCIC called GRIP. GRIP is a reasonably gradual type theory that addresses the issues above, featuring internal precision and general exception handling. For consistent reasoning about gradual terms, GRIP features an impure sort of types inhabited by errors and unknown terms, and a pure sort of strict propositions. By adopting a novel interpretation of the unknown term that carefully accounts for universe levels, GRIP satisfies graduality for a large and well-defined class of terms, in addition to being normalizing and a conservative extension of CIC. Internal precision supports reasoning about graduality within GRIP itself, for instance to characterize gradual exception-handling terms, and supports gradual subset types. We develop the metatheory of GRIP using a model formalized in Coq, and provide a prototype implementation of GRIP in Agda.[1]

1 INTRODUCTION

Extending gradual typing [Siek and Taha 2006; Siek et al. 2015] to dependent types is a challenging endeavor due to the intricacies of type checking and conversion in presence of imprecision at both the type and term levels. While early efforts looked at gradualizing specific aspects of a dependent type system (*e.g.,* subset types and refinements [Lehmann and Tanter 2017; Tanter and Tabareau 2015], or the fragment without inductive types [Eremondi et al. 2019]), recently Lennon-Bertrand et al. [2022] studied gradual typing in the context of the Calculus of Inductive Constructions (CIC), the theory at the core of many proof assistants such as Coq [The Coq Development Team 2020] and Agda [Norell 2007, 2009]. They uncover an inherent tension, dubbed the Fire Triangle of Graduality, which states that three fundamentally desirable properties cannot be fully satisfied simultaneously: (1) strong normalization, a property of particular relevance in the context of proof assistants, (2) conservativity with respect to CIC, namely the ability to faithfully embed the static theory in the

Preprint under submission.

[1] https://gitlab.inria.fr/kmaillar/grip-a-reasonably-gradual-type-theory

Authors' addresses: Kenji Maillard, Gallinette Project-Team, Inria, Nantes, France; Meven Lennon-Bertrand, Gallinette Project-Team, Inria, Nantes, France; Nicolas Tabareau, Gallinette Project-Team, Inria, Nantes, France; Éric Tanter, PLEIAD Lab, Computer Science Department (DCC), University of Chile, Santiago, Chile.

gradual theory,[2] and (3) graduality, which guarantees that typing and evaluation are monotone with respect to precision.[3]

Variants of Gradual CIC. Consequently, Lennon-Bertrand et al. [2022] develop a parametrized gradualization of CIC, called GCIC, which admits three variants, each sacrificing one property: GCIC$^{\mathcal{G}}$ satisfies both conservativity and graduality at the expense of admitting divergence, GCIC$^{\mathcal{N}}$ dynamically avoids non-termination but this carefulness inevitably leads to some terms violating graduality, and finally, GCIC$^{\uparrow}$ restricts the typing relation of CIC to exclude those non-gradual terms and hence does not admit all CIC terms. GCIC is a source language, whose semantics is given by elaboration to a dependently-typed calculus, called CastCIC, itself parametrized to account for the three variants. CastCIC is an extension of Martin-Löf type theory (MLTT) [Martin-Löf 1971] with (non-indexed) inductive types, and with exceptions as introduced by Pédrot and Tabareau [2018]. For a given type A, there are two exceptional terms, namely $\mathtt{err}_A$ representing runtime type errors, and $?_A$ representing the *unknown term*, which can optimistically stand for any term of type A. Additionally, CastCIC features a cast operator $\langle B \Leftarrow A \rangle\, t$, which supports treating a term t of type A as a term of type B, without requiring any relation between A and B. Intuitively, the cast operator is the identity when A and B are convertible, and fails when A and B are incompatible, for instance when A is the type of natural number and B is a function type. This intuition is made explicit by the notion of *(im)precision*: when a type A is more precise than B, written $A \sqsubseteq B$, then casting from A to B does not fail, and doing the roundtrip back to A is the identity; the formal formulation of this property, coined *graduality* by New and Ahmed [2018], is that when $A \sqsubseteq B$, the cast operations induce an embedding-projection pair between A and B. Additionally, $?$ is the least precise type, and therefore casting from A to the unknown type $?$ and back is always the identity. The maximality of the unknown type is a key element in the tension captured by the Fire Triangle of Graduality. Indeed, if $? \to ? \sqsubseteq ?$, then by graduality it is possible to to embed the untyped lambda calculus, and in particular the diverging term $\Omega := (\lambda\, x : ?.\, x\ x)\ (\lambda\, x : ?.\, x\ x)$.

Termination and universe levels. In GCIC, the unknown type is the unknown term at the universe type, $\square$. But due to predicativity in CIC there is in fact an infinite hierarchy of universes $\square_i$. This means that in GCIC there is one unknown type per level of the stratification; each $?_{\square_i}$ is the least precise type among all types at level i and below. The two GCIC variants that ensure termination avoid divergence by shifting universe levels either statically (GCIC$^{\uparrow}$) or dynamically (GCIC$^{\mathcal{N}}$). GCIC$^{\uparrow}$ restricts the typing rule of the function type compared to vanilla CIC by incrementing the universe level of the function type with respect to that of its components. Its main downside is that it is not a conservative extension of CIC: due to this modified typing rule, some perfectly valid CIC terms are statically rejected. The prototypical example is that of recursive large elimination, such as the type of n-ary functions over natural numbers (in Coq):

```
Fixpoint nArrow (n : ℕ) : □₀ := match n with 0 ⇒ ℕ | S m ⇒ ℕ → narrow m.
```

The term `nArrow n` is a type (*i.e.,* a term of type $\square_0$), and we have for example `nArrow 0` $\equiv \mathbb{N}$ and `nArrow 2` $\equiv \mathbb{N} \to \mathbb{N} \to \mathbb{N}$. The reason this definition is ill-typed in CIC$^{\uparrow}$ is that the universe level at which to define the resulting type is unbounded. Another more practical example is that of a dependently-typed `printf` function, whose actual arity depends on the input string.

In the context of a gradual proof assistant based on CIC, the normalizing and conservative variant GCIC$^{\mathcal{N}}$ is therefore the most appealing, as it ensures decidability of typing, (weak) canonicity,

[2] Not to be confused with logical conservativity!

[3] In the gradual typing literature, graduality is first known as the gradual guarantees [Siek et al. 2015]; the dynamic aspect thereof was later reformulated by New and Ahmed [2018] under a more semantic form, which turns out to be stronger than the dynamic gradual guarantee in the setting of dependent types [Lennon-Bertrand et al. 2022].

and supports all existing developments and libraries by virtue of being a conservative extension of CIC. GCIC$^{\mathcal{N}}$ avoids non-termination by introducing a universe shift during reduction, which unfortunately means that some terms break graduality. For instance, while `nArrow` is well-typed in GCIC$^{\mathcal{N}}$, the type `forall (n:ℕ), nArrow n` does not satisfy the embedding-projection property with respect to any unknown type $?_{\square_i}$, because the appropriate universe level is not known *a priori*. However, apart from the fact that GCIC$^{\mathcal{N}}$ does not satisfy graduality globally, little is known about its gradual properties as its metatheory in this regard has not been developed. In particular, there is no clear characterization of a class of terms for which graduality holds.

A Refined Stratification of Precision. In this work, we observe that by refining the stratification of precision we can develop a full account of graduality for an extension of CastCIC$^{\mathcal{N}}$, called GRIP. The key idea is that $?_{\square_i}$ should be the least precise type among all types at level i and below, *except* for dependent function types at level i (which are however still less precise than $?_{\square_{i+1}}$). We can precisely characterize problematic terms as those that are not *self-precise* (*i.e.,* more precise than themselves). As we will see, for function types, self-precision means monotonicity with respect to precision. A recursive large elimination as in `nArrow` is not monotone because, given $n \sqsubseteq ?_{\mathbb{N}}$, there is no fixed level i for which $\texttt{nArrow}\ n \sqsubseteq ?_{\square_i}$ for any n. We prove that the dynamic gradual guarantee holds in GRIP for any self-precise context, and that casts between types related by precision induce embedding-projection pairs between self-precise terms. Therefore, this shift in perspective in the interpretation of the unknown type and the associated notion of precision yields a gradual theory that conservatively extends CIC, is normalizing, and satisfies graduality for a large and well-defined class of terms.

Internalizing Precision, Reasonably. While we could study graduality for GRIP externally, we observe that we can exploit the expressiveness of the type-theoretic setting to internalize precision and its associated reasoning. In particular this makes it possible to state and prove, within the theory itself, results about (self-)precision and graduality for specific terms. Introducing internal precision in a gradual type theory however requires us to address two main obstacles:

(1) When adding exceptions to MLTT [Pédrot and Tabareau 2018], the theory becomes inconsistent as a logic, because it is possible to inhabit any type A by raising an exception $\texttt{err}_A$. In the gradual setting, there is also the alternative of using the unknown term $?_A$ to inhabit any type A. If we want to support internal reasoning about precision and graduality, we need to avoid these degenerate proofs and provide a logically consistent theory.
(2) The gradual type theory needs to satisfy extensionality principles in order to support the notion of precision as error approximation [New and Ahmed 2018]. Embracing extensionality principles in an intensional type theory such as MLTT or CIC is a challenge in itself.

We address both issues by combining recent advances in type theory: the reasonably exceptional type theory RETT [Pédrot et al. 2019] and the observational type theory TT$^{\text{obs}}$ [Pujet and Tabareau 2022]. First, RETT supports consistent reasoning about exceptional terms. It features a layer of possibly exceptional terms, and a separate layer of pure terms in which raising an exception is prohibited. This way, the consistency of the logical layer is guaranteed, while allowing non-trivial interaction with the exceptional layer. Technically, the two layers are defined using two distinct universe hierarchies. Second, based on the seminal work on Observational Type Theory [Altenkirch et al. 2007], TT$^{\text{obs}}$ provides a setoidal equality in a specific universe $\mathbb{P}$ of definitionally proof-irrelevant propositions. This universe of strict propositions, introduced by Gilbert et al. [2019] and supported in recent versions of Coq and Agda, makes it possible to define an extensional notion of equality, while trivializing the so-called higher coherence hell by imposing that any two proofs of a given equality are *definitionally* equal. The resulting theory is arguably much simpler and closer

to the current practice of proof assistants than cubical type theory [Cohen et al. 2015; Vezzosi et al. 2019], which is another approach to provide extensional principles with computational content.

A major insight of this work is to realize that we can actually merge the logical universe of RETT used to reason about exceptional terms with the universe $\mathbb{P}$ of proof-irrelevant propositions in order to define an internal notion of precision that is extensional and whose proofs cannot be trivialized with exceptional terms.

Applications of Internal Precision. In addition to supporting reasoning about the graduality of terms in a theory that is not globally gradual, internal precision makes it possible to support gradual subset types, in which a type can be refined by a proposition expressed using precision. Moreover, in the literature, exception handling is never considered when proving graduality because this mechanism inherently allows terms that do not behave monotonically with respect to precision. Internal precision enables us to support exception handling in the impure layer of the type theory, and to consistently reason about the graduality (or not) of exception-handling terms.

Structure of the paper. We propose GRIP, a novel gradual type theory with internal precision and a two-layer architecture that enables consistent reasoning about potentially failing and imprecise gradual programs. GRIP is a strongly-normalizing extension of CIC which satisfies graduality for a large and well-defined class of terms. After a brief informal overview of the main elements of GRIP and their applications (§2), we formalize GRIP as an extension of CastCIC with a sort of propositions (§3) and a precision relation for internal reasoning about graduality (§4). We present a model of GRIP in CIC, which validates its metatheoretical properties (§5). §6 discusses extensions and implementations of GRIP and §7 reviews related work. The Coq formalization of the model and the proof-of-concept implementation in Agda are provided at https://gitlab.inria.fr/kmaillar/grip-a-reasonably-gradual-type-theory.

2 A BRIEF OVERVIEW OF GRIP

CastCIC has been introduced by Lennon-Bertrand et al. [2022] as a variant of CIC with exceptional terms and a cast operator, designed to support the source gradual type theory GCIC. Due to the use of conversion for typing in dependently-typed systems, GCIC requires elaboration into CastCIC for both its static and dynamic semantics. This elaboration, which introduces casts as necessary to account for imprecision in GCIC terms, is not the focus of this work; instead, we tackle issues at the level of the design and semantics of the type theory with casts, CastCIC. After a quick refresher on CastCIC, this section introduces the two-layer architecture of GRIP for consistent reasoning about gradual programs, the notion of internal precision and its application to reason about graduality, including in the presence of exception handling, and gradual subset types.

2.1 Background on CastCIC

Technically, CastCIC features an impure hierarchy of sort of types $\square_i$ where one can freely use unknown terms (noted $?_A$ for any type A) and errors (noted $\mathtt{err}_A$). The hierarchy $\square_i$ is explicitly cumulative, meaning that there is a constructor $\iota : \square_i \to \square_{i+1}$ that permits to consider a type at level i as a type at level $i+1$. CastCIC also features inductive types such as natural numbers (noted $\mathbb{N}$), booleans (noted $\mathbb{B}$) and lists of elements of type A (noted $\mathbb{L}\ A$). The only difference with the corresponding inductive types in CIC is that there are two additional constructors for each inductive type, one corresponding to errors $\mathtt{err}$ and the other to the unknown term $?$ at that type. Additionally, CastCIC features casts, whose typing rule is

$$\frac{\Gamma \vdash A : \square_i \qquad \Gamma \vdash B : \square_i \qquad \Gamma \vdash t : A}{\Gamma \vdash \langle B \Leftarrow A \rangle\, t : B}$$

A cast converts any term of type A to a term of type B, with no constraint between A and B. This means that a cast propagates deeper when types are compatible, *e.g.,* two function types:

$$\langle A_2 \to B_2 \Leftarrow A_1 \to B_1 \rangle\, f \quad \rightsquigarrow \quad \lambda\, y : A_2.\langle B_2 \Leftarrow B_1 \rangle\, (f\ \langle A_1 \Leftarrow A_2 \rangle\, y)$$

But when A and B are not compatible, a cast reduces to an error in B, *e.g.,* between booleans and natural numbers, we have $\langle \mathbb{N} \Leftarrow \mathbb{B} \rangle\, \mathtt{true} \rightsquigarrow \mathtt{err}_{\mathbb{N}}$.

The main features of CIC that are absent in CastCIC are the presence of an impredicative universe of propositions and a general notion of indexed inductive types.

2.2 A Universe for Logical Reasoning

Directly inspired by the work on the reasonably exceptional type theory [Pédrot et al. 2019], GRIP features two distinct kind of sorts: the impure hierarchy of types $\square_i$ of CastCIC, and a pure impredicative sort of definitionally proof-irrelevant propositions $\mathbb{P}$. While propositions can be *about* gradual terms and errors, they cannot be themselves inhabited by unknown terms or errors, thereby ensuring consistent logical reasoning.

Lennon-Bertrand et al. [2022] show that no good notion of equality can be defined in the impure hierarchy of types because of an unsolvable tension between canonicity and the reduction of cast on equality. In GRIP, the absence of exceptions in $\mathbb{P}$ means the cast operator cannot be defined between propositions, and therefore the tension disappears. Consequently, $\mathbb{P}$ features a notion of equality $x =_A y$ for any terms x and y of type A similar to the observational equality recently introduced by Pujet and Tabareau [2022],[4] together with the corresponding transport operation:

$$\textsc{Transport}\quad \frac{\Gamma \vdash A : \square_i \qquad \Gamma \vdash B : \square_i \qquad \Gamma \vdash e : A = B \qquad \Gamma \vdash t : A}{\Gamma \vdash \mathtt{transport}\ A\ B\ e\ t : B}$$

Intuitively, transport can be seen as the *safe* version of cast, using a proof of equality between types in the logical layer as a guard to ensure it never fails.

To be able to reason about properties of inductive types in $\mathbb{P}$, their elimination principles needs to be extended for predicates in $\mathbb{P}$. However, contrarily to predicates valued in the impure hierarchy of types, there is no default behavior for errors and **?**. Thus eliminators in $\mathbb{P}$ require additional arguments to deal with those two exceptional cases, in a way reminiscent of try-catch for exception handling. For instance, the eliminator for $\mathbb{B}$ (if-then-else) is given by:

$$\mathtt{catch}^{\mathbb{P}}_{\mathbb{B}} : \forall (P : \mathbb{B} \to \mathbb{P}), P\ \mathtt{true} \to P\ \mathtt{false} \to P\ \mathtt{err}_{\mathbb{B}} \to P\ ?_{\mathbb{B}} \to \forall (b : \mathbb{B}), P\ b$$

In this logical layer, it becomes possible to reliably prove properties, because it is not possible to prove a false result in $\mathbb{P}$ by means of the unknown (or error) term, contrarily to $\square$. For instance, we can prove that casting from $\mathbb{B}$ to $\mathbb{N}$ is always an error, stated as

$$\forall (b : \mathbb{B}), \langle \mathbb{N} \Leftarrow \mathbb{B} \rangle\, b = \mathtt{err}_{\mathbb{N}}$$

This result is proven by a direct use of reflexivity of equality because the cast simply reduces to an error.

2.3 Internal Precision

GRIP features internal precision as an heterogeneous relation in the pure logical universe $\mathbb{P}$, defined between gradual types and terms of gradual types, as expressed by the typing rules:

$$\frac{\Gamma \vdash A, B : \square_i}{\Gamma \vdash A \sqsubseteq_i B : \mathbb{P}} \qquad \frac{\Gamma \vdash A, B : \square_i \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash t\ {}_A\sqsubseteq_B\ u : \mathbb{P}}$$

[4] This equality satisfies functional extensionality and uniqueness of identity proofs, but this is not crucial in this paper.

Because the universe level at which gradual types are defined plays a central role in the definition of precision, we explicitly annotate type precision with the level at which it occurs. Note that precision on proofs of propositions is undefined because there is simply no way to be imprecise in the logical layer.

Garcia et al. [2016] describe a systematic approach to design gradual languages, in which precision follows from the interpretation of gradual types as the set of static types that they denote. For instance, the type $\mathbb{N} \to ?$ denotes all function types with $\mathbb{N}$ as domain; this type is deemed more precise than the unknown type $?$ because the latter denotes any type. Therefore, precision among types coincides with the set inclusion of their denotations. Of course, in the context of a stratified hierarchy of types, with full dependency, the situation is more challenging.

To better reflect the semantics of CastCIC$^{\mathcal{N}}$ with respect to universe levels during reduction, which avoids diverging terms such as Ω without affecting typing, in GRIP we adjust the denotation of the unknown type at universe level i, $?_{\square_i}$, so that it excludes dependent function types at level i. Consequently, at level i, all type constructors except functions are more precise than $?_{\square_i}$, so the following propositions hold (mentioning only lists as the prototypical example of inductive types):

$$\square_i \sqsubseteq_{i+1} ?_{\square_{i+1}} \qquad \mathbb{L}\ A \sqsubseteq_i ?_{\square_i} \text{ whenever } A \sqsubseteq_i ?_{\square_i} \qquad \iota\, A \sqsubseteq_{i+1} ?_{\square_{i+1}} \qquad ?_{\square_i} \sqsubseteq_i ?_{\square_i}$$

In particular, in order to be more precise than the unknown type, a dependent function type needs to be *guarded* by an explicit use of cumulativity with $\iota : \square_i \to \square_{i+1}$. This means that we can derive $\iota(\mathbb{N} \to \mathbb{N}) \sqsubseteq_1 ?_{\square_1}$ and $\iota(?_{\square_0} \to ?_{\square_0}) \sqsubseteq_1 ?_{\square_1}$, but $\mathbb{N} \to \mathbb{N} \not\sqsubseteq_0 ?_{\square_0}$ and $?_{\square_0} \to ?_{\square_0} \not\sqsubseteq_0 ?_{\square_0}$.

Once the definition of precision on the unknown type is fixed, the rest of the definition is naturally obtained from congruence/extensional rules. We do not detail here the definition of internal term precision (presented in §4) but, for instance, precision between two functions $f\ {}_{\forall a,B\ a}{\sqsubseteq}_{\forall a',B'\ a'}\ g$ boils down to pointwise precision: $\forall a\, a'.\ a\ {}_{A}{\sqsubseteq}_{A'}\ a' \to f\ a\ {}_{B\ a}{\sqsubseteq}_{B'\ a'}\ g\ a'$. The only remaining subtlety is the definition of term precision in the impure sort $\square_i$, as it should be connected to type precision, because terms of $\square_i$ are types. Precision on types, when seen as terms of the sort $\square_i$, is the restriction of type precision to types that are more precise than $?_{\square_i}$, *i.e.,* $A\ {}_{\square_i}{\sqsubseteq}_{\square_i}\ B$ corresponds to $A \sqsubseteq_i B \wedge B \sqsubseteq_i ?_{\square_i}$.

Consequently, GRIP has the global property that $?_A$ is maximal for *term precision* of any type A, even when A is $\square_i$, but $?_{\square_i}$ is not maximal for *type precision* at level i, so as to avoid the Fire Triangle, as explained in §1. Conversely, however, type precision is stable by product formations, *i.e.,* in the non-dependent case if $A \sqsubseteq_i A'$ and $B \sqsubseteq_i B'$ then $A \to B \sqsubseteq_i A' \to B'$. This is not the case for term precision, again because of the Fire Triangle and of the maximality of $?_\square$ as a term.

This design forces certain terms to be non-monotone, in particular those built using large elimination. Consider the type-level function $\mathtt{t}_0 := \lambda\ \mathtt{b} \Rightarrow \mathtt{if}\ \mathtt{b}\ \mathtt{then}\ \mathbb{N}\ \mathtt{else}\ \mathbb{N} \to \mathbb{N}$. We have $\mathtt{false} \sqsubseteq ?_{\mathbb{B}}$, but we do not have $\mathtt{t}\ \mathtt{false} \equiv \mathbb{N} \to \mathbb{N} \sqsubseteq ?_{\square_0}$. We can address the issue in this simple case by posing $\mathtt{t}_1 := \lambda\ \mathtt{b} \Rightarrow \mathtt{if}\ \mathtt{b}\ \mathtt{then}\ \iota\,\mathbb{N}\ \mathtt{else}\ \iota\,(\mathbb{N} \to \mathbb{N})$, which explicitly uses cumulativity, so $\mathtt{t}_1$ is monotone as a function of type $\mathbb{B} \to \square_1$. Using cumulativity however does not work for recursive large elimination as the `nArrow` function discussed in the introduction. In that case, `nArrow` simply cannot be made monotone.

Armed with these notions of precision, it becomes possible to axiomatize directly in $\mathbb{P}$ the various properties they satisfy and their relation to casts. Note that because this axiomatization occurs in the definitionally proof-irrelevant universe $\mathbb{P}$, there is no need to endow the axioms with any computational meaning: they just need to be justified by a model to guarantees consistency (§5).

2.4 Internal Reasoning about Graduality

Graduality [New and Ahmed 2018] and the dynamic gradual guarantee (DGG) [Siek et al. 2015] are usually established as global properties of a gradual language. However, as mandated by the

Fire Triangle of Graduality [Lennon-Bertrand et al. 2022], graduality cannot hold globally in a terminating gradual extension of CIC. While Lennon-Bertrand et al. [2022] simply do not attempt to study graduality for CastCIC$^{\mathcal{N}}$, the situation of GRIP in this regard is both novel and unique: because precision is an internal notion within a type theory that allows for consistent reasoning, we can account for graduality. We can also exactly state the DGG theorem that holds in GRIP.

Dynamic Gradual Guarantee. In essence, the DGG says that if a term x is more precise than a term y, then for any evaluation context C, $C\ x$ "error approximates" $C\ y$—meaning that $C\ x$ can fail more than $C\ y$, but if it does not fail, then both are equivalent. Essentially, this property is about the *monotonicity* of contexts with respect to precision. In our setting, an evaluation context is simply a function from a term of some type A to a boolean $\mathbb{B}$, so the DGG is simply the monotonicity of functions, that is, $\mathtt{DGG} : \forall (A : \square)(C : A \to \mathbb{B})(x\ y : A), x\ {}_A\sqsubseteq_A\ y \to C\ x\ {}_{\mathbb{B}}\sqsubseteq_{\mathbb{B}}\ C\ y$.

As we have seen above with `nArrow`, not all functions are monotone in GRIP. To establish monotonicity internally in a general manner, we need a notion that not only makes sense for function types. Fortunately, a direct consequence of the pointwise definition of precision on functions is that monotonicity of functions corresponds to their *self-precision*. In general, we write $a^{\sqsubseteq_A}$ for self-precision, meaning that $a : A$ is such that $a\ {}_A\sqsubseteq_A\ a$.

In GRIP, $\mathtt{DGG}\ A\ C$ is equivalent to $C^{\sqsubseteq_{A\to\mathbb{B}}}$. In other words, for any type A and for any context C that is self-precise, we have the usual dynamic gradual guarantee between two elements x and y related by the precision over A. This means that we can understand existing gradual systems in which the DGG holds globally as systems where every context is self-precise by construction.

Graduality. Graduality as introduced by New and Ahmed [2018] is defined as the fact that when $A \sqsubseteq_i B$, then for any $a : A$ and $b : B$, there is an adjunction

$$\langle B \Leftarrow A\rangle\ a\ {}_B\sqsubseteq_B\ b \quad \leftrightarrow \quad a\ {}_A\sqsubseteq_B\ b \quad \leftrightarrow \quad a\ {}_A\sqsubseteq_A\ \langle A \Leftarrow B\rangle\ b.$$

and furthermore, the roundtrip is the identity on A: $\langle A \Leftarrow B\rangle\ \langle B \Leftarrow A\rangle\ a\ {}_A\sqsubseteq_A\ a$.

As we show in §4.2, GRIP globally satisfies graduality, except for the fact that $a : A$ and $b : B$ must both be self-precise for it to hold.

These properties can be used in several ways. A first potential use is to develop internally the theory of precision, showing for instance that casts between types related by precision do compose (which is not the case for arbitrary types). Another possible use is to derive proofs of precision on open terms that can appear during reasoning. For instance, when using gradual subset types (introduced in §2.6 below) to define functions, it becomes necessary to discharge proof obligations related to the precision of terms containing free variables.

2.5 Exception Handling and Graduality

All languages in the theoretical literature that address graduality are devoid of exception handling mechanisms. The reason is that handling runtime type errors makes it possible to define terms that are not monotone with respect to precision, and so graduality cannot hold globally. However, in practice, exception handling (and other language mechanisms in tension with graduality) are key ingredients and one would ideally like to account for them. As explained above, the situation of GRIP in this regard is new and singular: since we can internally and consistently reason about precision, we can support exception handling terms, and still establish their monotonicity as specific theorems proven in the type theory itself. Below we illustrate such an exception-handling term and its proof of monotonicity within GRIP.

The catch operator on $\mathbb{B}$ is not monotone with respect to precision. Consider its type signature:

$$\mathtt{catch}^{\square}_{\mathbb{B}} : \forall (A : \square)\ (a_{\mathtt{true}} : A)\ (a_{\mathtt{false}} : A)\ (a_{\mathtt{err}_{\mathbb{B}}} : A)\ (a_{?_{\mathbb{B}}} : A), \mathbb{B} \to A$$

There is no reason for the value $a_{\mathtt{err}_{\mathbb{B}}}$ given when catching an error to be more precise than the values $a_{\mathtt{true}}$ and $a_{\mathtt{false}}$. In our setting, the catch operation (and its dependent generalization) can perfectly be considered, and we can show in specific use of catch that precision is preserved.

To illustrate, consider the following optimized implementation of (iterative) multiplication of a list of natural numbers, with two functions, that takes advantage of the fact that 0 is an absorbing element (we use pattern matching syntax for induction on lists to ease the reading):

$$\begin{array}{lll} \mathtt{mult}^{\mathtt{err}}_{\mathbb{L}} \ \mathtt{nil} & := & 1 \\ \mathtt{mult}^{\mathtt{err}}_{\mathbb{L}} \ (\mathtt{cons}\ n\ l) & := & \mathtt{if}\ (\mathtt{is\_zero}\ n)\ \mathtt{then}\ \mathtt{err}_{\mathbb{N}}\ \mathtt{else}\ n * \mathtt{mult}^{\mathtt{err}}_{\mathbb{L}} l \\ \mathtt{mult}_{\mathbb{L}} \ l & := & \mathtt{catch}^{\Box}_{\mathbb{N}}\ \mathbb{N}\ 0\ (\lambda\, n : \mathbb{N}.n)\ 0\ ?_{\mathbb{N}}\ (\mathtt{mult}^{\mathtt{err}}_{\mathbb{L}}\ l) \end{array}$$

The first function $\mathtt{mult}^{\mathtt{err}}_{\mathbb{L}}$ returns an error as soon as a 0 is encountered in the list, short-circuiting the recursive computation. The wrapper function $\mathtt{mult}_{\mathbb{L}}$ catches errors raised by $\mathtt{mult}^{\mathtt{err}}_{\mathbb{L}}$ and returns 0 in that case. The function $\mathtt{mult}_{\mathbb{L}}$ is not monotone in general because when the input list is an error, it returns the value 0, which is not more precise than the return value on other lists. But $\mathtt{mult}_{\mathbb{L}}$ is monotone on lists that do not contain errors, because in such cases errors are used in a delimited manner in order to optimize execution. In GRIP, we can make this explicit and prove the following theorem:

$$\mathtt{mult}^{\sqsubseteq}_{\mathbb{L}} : \forall (l\ l' : \mathbb{L}\ \mathbb{N}), \mathtt{not\text{-}err}_{\mathbb{L}}\ l \to l\ {}_{\mathbb{L}\ \mathbb{N}}{\sqsubseteq}_{\mathbb{L}\ \mathbb{N}}\ l' \to \mathtt{mult}_{\mathbb{L}}\ l\ {}_{\mathbb{N}}{\sqsubseteq}_{\mathbb{N}}\ \mathtt{mult}_{\mathbb{L}}\ l'.$$

where $\mathtt{not\text{-}err}_{\mathbb{L}}$ is a predicate ensuring that the list in not $\mathtt{err}_{\mathbb{L}\ \mathbb{N}}$ and does not contains $\mathtt{err}_{\mathbb{N}}$ in its elements. A comprehensive description of this example is given in §6.4.

2.6 Gradual Subset Types

The logical layer $\mathbb{P}$ enables stating and proving formal properties on the gradual, impure layer $\Box$. But in a dependently-typed setting, it is also important to be able to use the properties stated in $\mathbb{P}$ to constrain types in $\Box$, using for instance *subset types*. Recall that a subset type is a type A enriched with a proposition P, noted $\{a : A \,\&\, P\ a\}$, and an inhabitant is a dependent pair $(a; p)$, such that $a : A$ and $p : P\ a$. This means that in GRIP we need a way to embed $\mathbb{P}$ into $\Box$. Note that this cannot be a direct injection, as propositions in $\mathbb{P}$ cannot be inhabited with exceptions. Therefore, we need a special operator $\mathbb{B}\mathsf{ox} : \mathbb{P} \to \Box$ that takes a proposition P and freely adds $\mathtt{err}_{\mathbb{B}\mathsf{ox}\ P}$ and $?_{\mathbb{B}\mathsf{ox}\ P}$ to P. This allows us to define lists of size n as the type

$$\mathsf{Sized}\mathbb{L}\ A\ n := \{l : \mathbb{L}\ A \,\&\, \mathbb{B}\mathsf{ox}\ (\mathsf{len}\ l = n)\}.$$

This way, we can gradually define the $\mathsf{append}_?$ function as

$$\begin{array}{l} \mathsf{append}_? : \forall A\ n\ m,\ \mathsf{Sized}\mathbb{L}\ A\ n \to \mathsf{Sized}\mathbb{L}\ A\ m \to \mathsf{Sized}\mathbb{L}\ A\ (n+m) \\ \mathsf{append}_?\ A\ n\ m\ (l; \_)\ (l'; \_) := (l +\!\!+ l'; ?_{\mathbb{B}\mathsf{ox}\ (\mathsf{len}\ (l +\!\!+ l') = n+m)}) \end{array}$$

where the proof that the result is of the right size is avoided through imprecision. It is also possible to define the precise append function that contains the actual proof that the resulting size is valid:

$$\begin{array}{l} \mathsf{append} : \forall A\ n\ m,\ \mathsf{Sized}\mathbb{L}\ A\ n \to \mathsf{Sized}\mathbb{L}\ A\ m \to \mathsf{Sized}\mathbb{L}\ A\ (n+m) \\ \mathsf{append}\ A\ n\ m\ (l; \mathsf{box}\ p)\ (l'; \mathsf{box}\ p') := (l +\!\!+ l'; +\!\!+\mathtt{lemma}\ l\ l' \cdot \mathsf{ap}_2 + p\ p') \end{array}$$

where $+\!\!+\mathtt{lemma}$ is the proof that the length of two appended lists is equal to the sum of their lengths, $e \cdot e'$ is the concatenation of equality and ap_2 is a witness that (binary) functions preserve equalities.

Given these two definitions of append, what is the property satisfied by the precise append function that is not satisfied by the imprecise one? In GRIP, these two functions can be distinguished

in the logical layer by using the following predicate, which indicates that a property in the impure layer has *really* been proven:

$$\begin{aligned}&\mathtt{valid}_{\mathbb{B}\mathtt{ox}} : \forall P : \mathbb{P},\ \mathbb{B}\mathtt{ox}\ P \to \mathbb{P}\\ &\mathtt{valid}_{\mathbb{B}\mathtt{ox}}\ P\ (\mathtt{box}\ p) \coloneqq \top\\ &\mathtt{valid}_{\mathbb{B}\mathtt{ox}}\ P\ (\mathtt{err}_{\mathbb{B}\mathtt{ox}\ P}) \coloneqq \bot\\ &\mathtt{valid}_{\mathbb{B}\mathtt{ox}}\ P\ (?_{\mathbb{B}\mathtt{ox}\ P}) \coloneqq \bot\end{aligned}$$

Posing $\mathtt{valid}_{\mathtt{Sized}\mathbb{L}}(\_; p) \coloneqq \mathtt{valid}_{\mathbb{B}\mathtt{ox}}\ \_\ p$, the precise append function is the only one of the two versions for which one can prove:

$$\mathtt{valid\_append} : \forall\ A\ n\ m\ l\ l', \mathtt{valid}_{\mathtt{Sized}\mathbb{L}}\ l \to \mathtt{valid}_{\mathtt{Sized}\mathbb{L}}\ l' \to \mathtt{valid}_{\mathtt{Sized}\mathbb{L}}(\mathtt{append}\ A\ n\ m\ l\ l')$$

In a gradual setting, we can also use the unknown term in order to avoid an explicit definition of the resulting size of the list. For instance, the filter function can be given the imprecise type

$$\mathtt{filter} : \forall A\ n\ (P : A \to \mathbb{P}),\ \mathtt{Sized}\mathbb{L}\ A\ n \to \mathtt{Sized}\mathbb{L}\ A\ ?_{\mathbb{N}}$$

However, there is no way to give a valid implementation of a filter function of that type, because the size of the filtered list cannot be proven to be equal to $?_{\mathbb{N}}$ in the logical layer. Taking advantage of the internal notion of precision, we can define an alternative notion of sized list in GRIP as

$$\mathtt{Sized}\mathbb{L}_{\sqsubseteq}\ A\ n \coloneqq \{l : \mathbb{L}\ A\ \&\ \mathbb{B}\mathtt{ox}\ (\mathtt{len}\ l\ {}_{\mathbb{N}}\!\sqsubseteq_{\mathbb{N}}\ n)\}.$$

Using this notion of sized lists, it is possible to define a valid filter function of type

$$\mathtt{filter}_{\sqsubseteq} : \forall A\ n\ (P : A \to \mathbb{P}),\ \mathtt{Sized}\mathbb{L}_{\sqsubseteq}\ A\ n \to \mathtt{Sized}\mathbb{L}_{\sqsubseteq}\ A\ ?_{\mathbb{N}}.$$

because the proof that the size of the filtered list is more precise than $?_{\mathbb{N}}$ directly follows from the fact that $?_{\mathbb{N}}$ is the maximal element of type $\mathbb{N}$.

3 GRADUAL TYPES AND PURE PROPOSITIONS

In this section, we present the two-layer core of GRIP, intended to be both a gradual cast calculus, target for elaboration of a gradual surface language, and a pure language to consistently reason about programs in that cast calculus. In § 3.1, we give an overview of the gradual part of the language, while §3.2 introduces the pure sort of propositions. Finally, §3.3 discusses how to soundly support interactions between these two layers.

3.1 The Impure Layer of Gradual Terms

As seen in §2.1, CastCIC is an extension of MLTT with primitives for gradual typing, namely casts, errors and unknown terms. For the impure layer of gradual terms, GRIP is basically CastCIC [Lennon-Bertrand et al. 2022], with some minor modifications and presentation differences highlighted below, in particular the support for exception handling and explicit cumulativity.

The syntax and typing rules of the gradual layer of GRIP are given in Fig. 1. They feature a hierarchy of universes $\square_i$, dependent products Π introduced by λ-abstraction and destructed by applications, and inductive types, introduced by constructors and destructed by catch operators. Here we do not consider inductive types with indices, such as equality, whose treatment is deferred to §6.1. For readability we only formally present lists $\mathbb{L}$, however the calculus can readily be extended with other parametrized instances of $\mathbb{W}$-types, as done for CastCIC. Throughout the article, and in particular for examples, we take the liberty to use dependent sums Σ, natural numbers $\mathbb{N}$ and booleans $\mathbb{B}$. The typing rule (List-Catch) for the catch operator on lists requires two additional arguments with respect to the usual recursor on lists, one for the case of an error, and one for **?**. Note that the usual recursor on lists $\mathtt{ind}^{\square}_{\mathbb{L}}$ which simply propagates `err` and **?**, as used in CastCIC, can be recovered from the catch operator by defining $h_{\mathtt{err}}$ to be `err` and $h_?$ to be **?**.

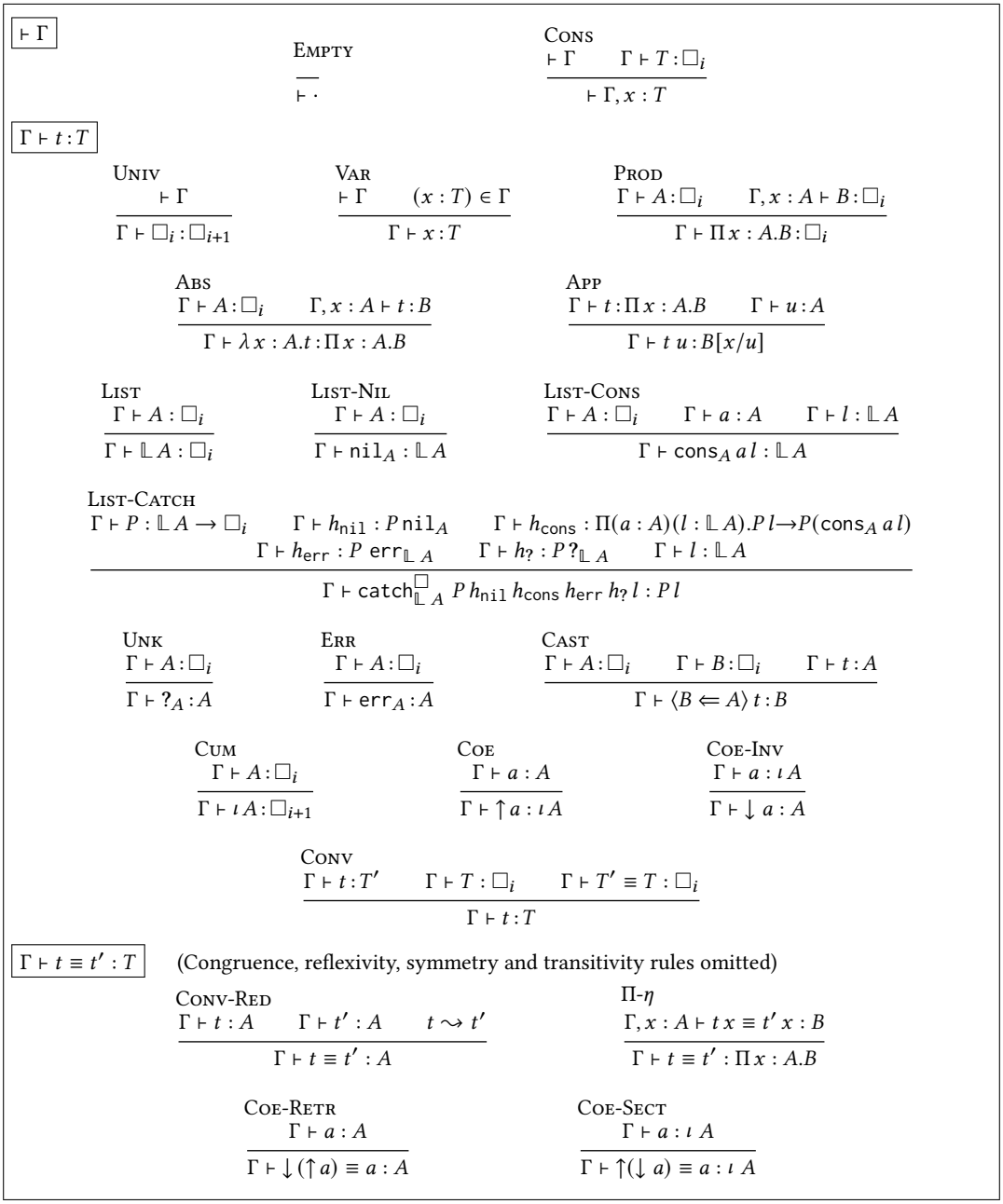

Fig. 1. GRIP: typing of the impure layer — based on CastCIC

GRIP uses explicit cumulativity (as in Agda), in the form of an operator ι lifting a type from one universe to the next, and coercions $\uparrow$ and $\downarrow$ between a type and its lift. We choose explicit cumulativity due to the central role it plays in the definition of internal precision (§4—see §6 for further discussion on explicit versus implicit cumulativity). As for the gradual part of the calculus,

it features the unknown terms $?_A : A$, errors $\mathtt{err}_A : A$, and casts $\langle B \Leftarrow A \rangle\ a$ between arbitrary types at the same universe level.

As any dependent type theory, GRIP relies on a notion of conversion that allows us to convert a term of type T' to a term of type T (Rule Conv) as soon the two types are convertible. Conversion is defined as the reflexive, symmetric and transitive closure of reduction with the additional η-conversion for functions and the fact that $\uparrow$and $\downarrow$ are inverse of each other.

The dynamic behavior of these terms is presented by means of a reduction relation in Fig. 2, directly adapted from that of CastCIC. There are three sets of rules. The first is for standard rules of MLTT, *i.e.,* the usual β-rule for functions and ι-rule for lists. The second corresponds to propagation of both ? and err as exceptions as advocated for by Pédrot and Tabareau [2018]. The last describes the behavior of the cast primitive, which computes based on the shape of its two type arguments. The first five rules propagate casts between types with the same head constructor. The next four correspond to failures, either when the source and target types are incompatible, when one of them is an error, or when trying to cast a product type into the unknown type of its level. This last rule Π-Err is crucial for normalization, as it is responsible for the failure of terms such as Ω. Next, rule Up-Down can be understood as a form of ι-rule for $?_\square$: it showcases the fact that casts into $?_\square$ work as canonical forms for it (when their domain is of a certain form), with casts from $?_\square$ as destructors. Finally rule $\mathbb{L}$-Dec decomposes casts from a list into the unknown type through $\mathbb{L}?_\square$, the most general type with $\mathbb{L}$ as a head constructor, letting rules Catch-nil and Catch-cons further decompose the innermost cast if applicable.

3.2 The Pure Layer for Reasoning on Gradual Terms

The casts $\langle B \Leftarrow A \rangle\ t$ and exceptional terms $\mathtt{err}_A, ?_A$ are fundamental features to enable gradual programming. However, as a consequence all types are inhabited, so logical consistency, and thus meaningful internal reasoning on programs, is lost. To remedy this problem, following the insight of RETT [Pédrot et al. 2019], we introduce an additional layer dedicated to sound reasoning, which must therefore be free of the gradual primitives. As in RETT, the separation between the impure and pure layers is controlled by means of sorts: alongside the impure hierarchy of gradual terms $\square_i$, we introduce a new impredicative[5] sort $\mathbb{P}$ of definitionally proof-irrelevant pure propositions.

In more details, Fig. 3 shows how GRIP extends what was essentially CastCIC with this new sort $\mathbb{P}$ ($\mathbb{P}$-Wf). In particular, an extension of conversion specifies that any two proofs of the same proposition are convertible ($\mathbb{P}$-Irr). We use $\mathbb{s}$ for a generic sort, that is either $\mathbb{P}$ or $\square_i$ for some i. At this stage, there are only two ways to construct propositions. On one side, the empty proposition $\bot$ ($\bot$-Wf) with no introduction, and elimination in the form of an explosion principle ($\bot$-Elim). On the other, universal quantification over propositions or types ($\forall$-Wf) introduced by λ-abstraction ($\forall$-Intro) and eliminated by application ($\forall$-Elim). Implication $P \to Q$ between propositions is defined as the non-dependent quantification $\forall(\_ : P).Q$. More interesting ones will be added later, such as the precision relation (Fig. 4). However, further logical connectives can already be encoded on top of the primitives we already have, using impredicativity and definitional proof-irrelevance [Gilbert et al. 2019]. For instance, the proposition true can defined by $\top := \bot \to \bot$.

The success of the separation between the layers is given by the following theorem, which will be proven in §5.

Theorem 1 (Logical soundness of GRIP). *If* MLTT *extended with strict propositions is consistent then there is no closed proof* $\vdash e : \bot$ *of the empty proposition* $\bot : \mathbb{P}$ *in* GRIP.

[5] Impredicativity is inessential but simplifies the exposition while matching the model in §5; the Agda development shows how this presentation can be adapted to a predicative hierarchy $\mathbb{P}_i$.

$\boxed{\text{Head}, \text{Whnf}_\square, \text{head} : \text{Whnf}_\square \to \text{Head}}$

$$\text{Head} \ni h ::= \square \mid \Pi \mid \mathbb{L} \mid \iota\, A \qquad \text{Whnf}_\square ::= \square_i \mid \Pi\, x : A.B \mid \mathbb{L}\, A \mid \iota\, A$$

$$\text{head}\,(\square_i) := \square \qquad \text{head}\,(\Pi\, x : A.B) := \Pi \qquad \text{head}\,(\mathbb{L}\, A) := \mathbb{L} \qquad \text{head}\,(\iota\, A) := \iota$$

$\boxed{t \rightsquigarrow t'}$

Standard MLTT

$$\Pi\text{-}\beta : (\lambda\, x : A.t)\, u \rightsquigarrow t[u/x] \qquad \textsc{Catch-nil} : \mathsf{catch}^{\square}_{\mathbb{L}\, A}\, P\, h_{\mathsf{nil}}\, h_{\mathsf{cons}}\, h_{\mathsf{err}}\, h_?\, \mathsf{nil}_A \rightsquigarrow h_{\mathsf{nil}}$$

$$\textsc{Catch-cons} : \mathsf{catch}^{\square}_{\mathbb{L}\, A}\, P\, h_{\mathsf{nil}}\, h_{\mathsf{cons}}\, h_{\mathsf{err}}\, h_?\, (\mathsf{cons}_A\, a\, l) \rightsquigarrow h_{\mathsf{cons}}\, a\, l\, (\mathsf{catch}^{\square}_{\mathbb{L}\, A}\, P\, h_{\mathsf{nil}}\, h_{\mathsf{cons}}\, h_{\mathsf{err}}\, h_?\, l)$$

Propagation rules for ? and err

$$\Pi\text{-}\textsc{Unk} : ?_{\Pi(x:A).B} \rightsquigarrow \lambda(x : A).?_B \qquad \Pi\text{-}\textsc{Err} : \mathsf{err}_{\Pi(x:A).B} \rightsquigarrow \lambda(x : A).\mathsf{err}_B$$

$$\textsc{Cum-Unk} : ?_{\iota\, A} \rightsquigarrow \uparrow ?_A \qquad \textsc{Catch-Unk} : \mathsf{catch}^{\square}_{\mathbb{L}\, A}\, P\, h_{\mathsf{nil}}\, h_{\mathsf{cons}}\, h_{\mathsf{err}}\, h_?\, ?_{\mathbb{L}\, A} \rightsquigarrow h_?$$

$$\textsc{Cum-Err} : \mathsf{err}_{\iota\, A} \rightsquigarrow \uparrow \mathsf{err}_A \qquad \textsc{Catch-Err} : \mathsf{catch}^{\square}_{\mathbb{L}\, A}\, P\, h_{\mathsf{nil}}\, h_{\mathsf{cons}}\, h_{\mathsf{err}}\, h_?\, \mathsf{err}_{\mathbb{L}\, A} \rightsquigarrow h_{\mathsf{err}}$$

$$\mathbb{L}\text{-}\textsc{Cast-Unk} : \langle \mathbb{L}\, A' \Leftarrow \mathbb{L}\, A'' \rangle\, ?_{\mathbb{L}\, A} \rightsquigarrow ?_{\mathbb{L}\, A'} \qquad \mathbb{L}\text{-}\textsc{Cast-Err} : \langle \mathbb{L}\, A' \Leftarrow \mathbb{L}\, A'' \rangle\, \mathsf{err}_{\mathbb{L}\, A} \rightsquigarrow \mathsf{err}_{\mathbb{L}\, A'}$$

$$\textsc{Down-Unk} : \langle T \Leftarrow ?_\square \rangle\, ?_{?_\square} \rightsquigarrow ?_T \qquad \textsc{Down-Err} : \langle T \Leftarrow ?_\square \rangle\, \mathsf{err}_{?_\square} \rightsquigarrow \mathsf{err}_T \qquad \text{when } T \in \text{Whnf}_\square$$

Reduction rules for cast

$$\Pi\text{-}\Pi : \langle \Pi(y : A_2).B_2 \Leftarrow \Pi(x : A_1).B_1 \rangle\, f \rightsquigarrow \lambda\, y : A_2.\langle B_2 \Leftarrow B_1[\langle A_1 \Leftarrow A_2 \rangle\, y/x] \rangle\, (f\, \langle A_1 \Leftarrow A_2 \rangle\, y)$$

$$\textsc{Cum-Cum} : \langle \iota\, A' \Leftarrow \iota\, A \rangle \uparrow t \rightsquigarrow \langle A' \Leftarrow A \rangle\, t \qquad \textsc{Univ-Univ} : \langle \square_i \Leftarrow \square_i \rangle\, A \rightsquigarrow A$$

$$\mathbb{L}\text{-}\mathbb{L}\text{-}\textsc{Nil} : \langle \mathbb{L}\, A' \Leftarrow \mathbb{L}\, A'' \rangle\, \mathsf{nil}_A \rightsquigarrow \mathsf{nil}_{A'}$$

$$\mathbb{L}\text{-}\mathbb{L}\text{-}\textsc{Cons} : \langle \mathbb{L}\, A' \Leftarrow \mathbb{L}\, A'' \rangle\, (\mathsf{cons}_A\, a\, l) \rightsquigarrow \mathsf{cons}_{A'}\, \left(\langle A' \Leftarrow A \rangle\, a\right)\, \left(\langle \mathbb{L}\, A' \Leftarrow \mathbb{L}\, A \rangle\, l\right)$$

$$\textsc{Head-Err} : \langle T' \Leftarrow T \rangle\, t \rightsquigarrow \mathsf{err}_{T'} \qquad \text{when } T, T' \in \text{Whnf}_\square \text{ and } \text{head}\, T \neq \text{head}\, T'$$

$$\textsc{Dom-Err} : \langle T \Leftarrow \mathsf{err}_\square \rangle\, t \rightsquigarrow \mathsf{err}_T \qquad \textsc{Cod-Err} : \langle \mathsf{err}_\square \Leftarrow T \rangle\, t \rightsquigarrow \mathsf{err}_{\mathsf{err}_\square} \qquad \text{when } T \in \text{Whnf}_\square$$

$$\textsc{Cast-}\Pi\textsc{-Err} : \langle ?_\square \Leftarrow \Pi\, x : A.B \rangle\, f \rightsquigarrow \mathsf{err}_{?_\square}$$

$$\textsc{Up-Down} : \langle X \Leftarrow ?_{\square_i} \Leftarrow Y \rangle\, t \rightsquigarrow \langle X \Leftarrow Y \rangle\, t \qquad \text{when } X \in \text{Whnf}_\square \text{ and } Y \text{ is } \mathbb{L}\, ?_\square, \square \text{ or } \iota\, A$$

$$\mathbb{L}\text{-}\textsc{Dec} : \langle ?_{\square_i} \Leftarrow \mathbb{L}\, A \rangle\, t \rightsquigarrow \langle ?_{\square_i} \Leftarrow \mathbb{L}\, ?_{\square_i} \Leftarrow \mathbb{L}\, A \rangle\, t \qquad \text{when } \mathbb{L}\, A \neq \mathbb{L}\, ?_{\square_i}$$

Fig. 2. GRIP: Reduction rules (Congruence closure rules omitted) – adapted from CastCIC

3.3 Crossing Sort Boundaries

Eliminations. Because of the important differences between the two layers of GRIP, their interactions need to be finely controlled in order to stay well-behaved. This is done by providing restricted elimination of inhabitants of types from one layer to types of the other.

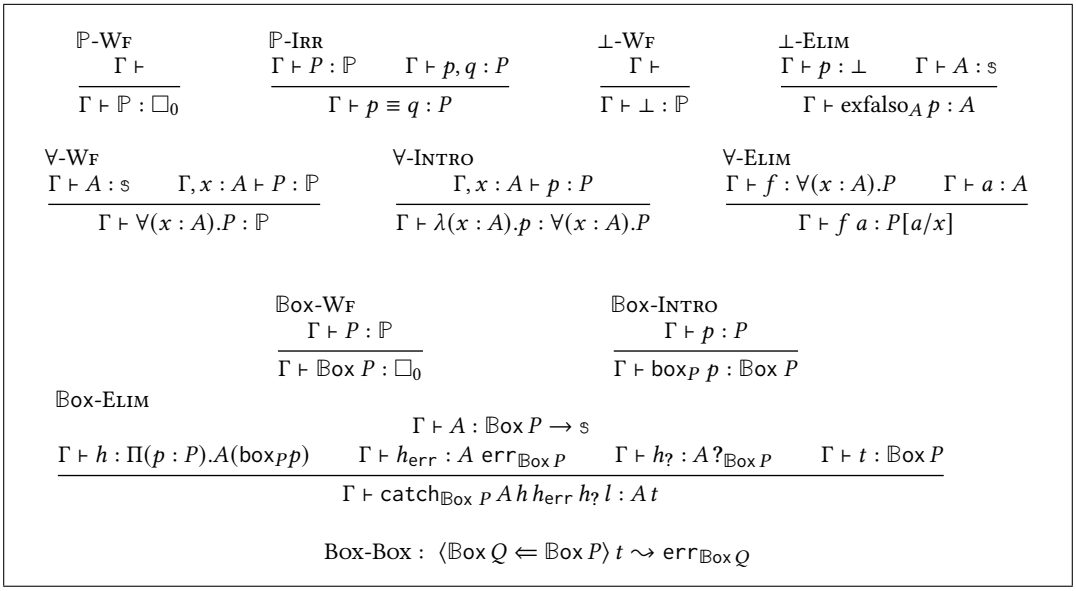

Fig. 3. GRIP: Extensions of typing and reduction for propositions and boxing ($\mathbb{s} = \mathbb{P}$ or $\Box_i$)

In one direction, eliminating from the pure propositional layer to the impure gradual one is allowed only through the empty proposition $\bot$, by using the explosion principle, a.k.a. *ex-falso* ($\bot$-Elim). This can be seen as a strengthening of the singleton elimination criterion of the usual `Prop` sort of Coq, in a way that respects definitional proof-irrelevance [Gilbert et al. 2019]. Effectively, one is allowed to use a proof of a proposition to inhabit a type only to show that we are in an inconsistent context, typically in an unreachable branch of a match. In practice, this ends up not being too restrictive, since quite a few propositions are defined on top of $\bot$. For instance, internal precision defined in §4 ultimately reduces to a combination of $\forall$ and $\bot$ after case analysis on its type parameters.

In the other direction, eliminators from the impure layer to the pure layer need to take errors and ? into account. Indeed, since these terms do not exist as propositions, they cannot be used when matching on an impure argument. Thus, the need for a `catch` recursor is even more dire than for types, because we cannot rely on errors in the target type to provide "default" values for an `err` or ? scrutinee, as an `ind` recursor does. On lists, for instance, we get $\mathtt{catch}^{\mathbb{P}}_{\mathbb{L}\ A}$, which behaves exactly the same as $\mathtt{catch}^{\Box}_{\mathbb{L}\ A}$ except that it can be used on predicates of type $\mathbb{L}\ A \to \mathbb{P}$.

Embedding Propositional Invariants within $\Box$. In order to quantify over a proposition in a type, or carry a proof along some data, propositions must be embeddable into types and equipped with `err` and ?. As illustrated in §2.6 with the case of gradual subset types, this is achieved through the type $\mathbb{B}\mathrm{ox}\ P$ (Fig. 3) that packs a proposition $P : \mathbb{P}$ ($\mathbb{B}$ox-Wf). A proof $p : P$ of a proposition can be used to inhabit $\mathbb{B}\mathrm{ox}\ P$ using the constructor $\mathtt{box}_P$ ($\mathbb{B}$ox-Intro). Moreover, as any other type, $\mathbb{B}\mathrm{ox}\ P$ is equipped with exceptional constructors $\mathtt{err}_{\mathbb{B}\mathrm{ox}\ P}$ and $?_{\mathbb{B}\mathrm{ox}\ P}$. The eliminator on $\mathbb{B}\mathrm{ox}$ is given by a catch operator, similar to the one for lists ($\mathbb{B}$ox-Elim), whose obvious reduction rules are omitted.

We extend the reduction of casts to $\mathbb{B}\mathrm{ox}$ ($\mathbb{B}$ox-Box) by reducing a cast between $\mathbb{B}\mathrm{ox}$-types to an error. This peculiar definition is chiefly due to the fact that we cannot decide entailment between arbitrary propositions P and Q, and so cannot decide when casting $\mathtt{box}_P\ p$ to $\mathbb{B}\mathrm{ox}\ Q$ should return some $\mathtt{box}_Q\ p'$ or fail.

4 INTERNALIZING PRECISION

The pure logical layer $\mathbb{P}$ is used to assert properties of the impure gradual layer $\square$. But none of the primitives introduced in §3 enable direct reasoning on the most important relation between gradual programs: precision. In this section, we provide exactly this, by extending the logical layer with an internal precision relation specifying the behavior of casts (§4.1).

However, having a definition of precision is not enough: as we cannot reason by induction on types, general properties such as transitivity of precision cannot be derived from the definition in §4.1 alone. This is why we also need to directly add properties of precision (§4.2). As those are added as new constants inhabiting propositions, we do not need to specify anything about them. Indeed, all inhabitants of propositions are definitionally equal, so none of them is better than another. The only thing of importance is to preserve consistency of the theory, by ensuring that the properties are validated by the model (§5).

Although the impure layer does not globally satisfy graduality, a large fragment of the language behaves well, in the sense that it is monotone with respect to precision (§4.3). In particular, we show that this fragment subsumes CastCIC$^{\uparrow}$, the normalizing gradual variant proposed by [Lennon-Bertrand et al. 2022].

4.1 The Precision Relation

The *raison d'être* of the propositional layer is to host the precision relation, that provides an entrypoint for specifying correctness properties of casts. Precision is formulated in two distinct flavours for types and terms: a homogeneous relation $A \sqsubseteq_i B$ on types $A, B : \square_i$ of a common universe level i, and a heterogeneous relation $a \ {}_A\sqsubseteq_B \ b$ between terms $a : A$ and $b : B$. The content of these two precision relations is described by their behaviour on their type parameters. In practice we present these relations through a confluent reduction system in Fig. 4, corresponding to a definition by case analysis on the type parameters, which is how the model of §5 proceeds.

Before diving into the actual content of these relations, let us explain the two main properties we expect to hold. First, the precision relation should be *transitive*: there should be an operation $\cdot$ such that if $e : A \sqsubseteq_i B$ and $e' : B \sqsubseteq_i C$ then $e \cdot e' : A \sqsubseteq_i C$. Second, the precision relation cannot be reflexive. Indeed, reflexivity at function types $A \to B$ entails monotonicity: due to the way we define precision, if a function $f : A \to B$ verifies $f \sqsubseteq f$ then for any $a \ {}_A\sqsubseteq_A \ a'$, $f\ a \ {}_B\sqsubseteq_B \ f\ a'$. But we do not want to globally forbid such non-monotone features, as we rather made the design choice to allow some non-monotonicity in GRIP, *e.g.* the `catch` construct. As a consequence, reflexivity becomes a property, and we say that a type $A : \square_i$ is *self-precise*, noted $A^{\sqsubseteq i}$, when it is a reflexive element of $\sqsubseteq_i$. Similarly, a term $a : A$ of a self-precise type $e : A^{\sqsubseteq i}$ is called *self-precise*, noted $a^{\sqsubseteq_A}$, when it is related to itself by $\ {}_A\sqsubseteq_A$. Not every type is self-precise, but the precision relation is still *quasi-reflexive*: whenever two types A, B are related by precision, both should be self-precise.[6], meaning that we have self-precision proofs $\lfloor e \rfloor : A^{\sqsubseteq i}, \lceil e \rceil : B^{\sqsubseteq i}$, so the precision relation is *quasi-reflexive*.

Let now turn to the actual content of the precision relations as defined in Fig. 4. The first two rules $\square$-$\sqsubseteq$ and $\mathbb{P}$-$\sqsubseteq$ tell when precision at the type and term levels is well-typed. $\square$-Refl-Ty next states that each universe $\square_i$ is self-precise (as a type), and heterogeneous precision between types reduces to homogeneous precision between types more precise than $?_{\square_i}$ by virtue of $\square$-$\sqsubseteq$, tying the knot between the two notions. As a consequence, a proof of precision $A \ {}_{\square_i}\sqsubseteq_{\square_i} \ A$ entails that $A \sqsubseteq_i A$ as well as $A \sqsubseteq_i ?_{\square_i}$. Precision at product types is the crux of the definition of precision, we defer its explanation of Π-Cong to after the other rules. For now, it is only important to note that contrarily to other type formers, there is no rule to relate product types *as terms*, only *as types*. This

[6]In order to obtain transitivity on function types, the precision relation needs to be at least co-transitive, a property obtained here as a consequence of quasi-reflexivity.

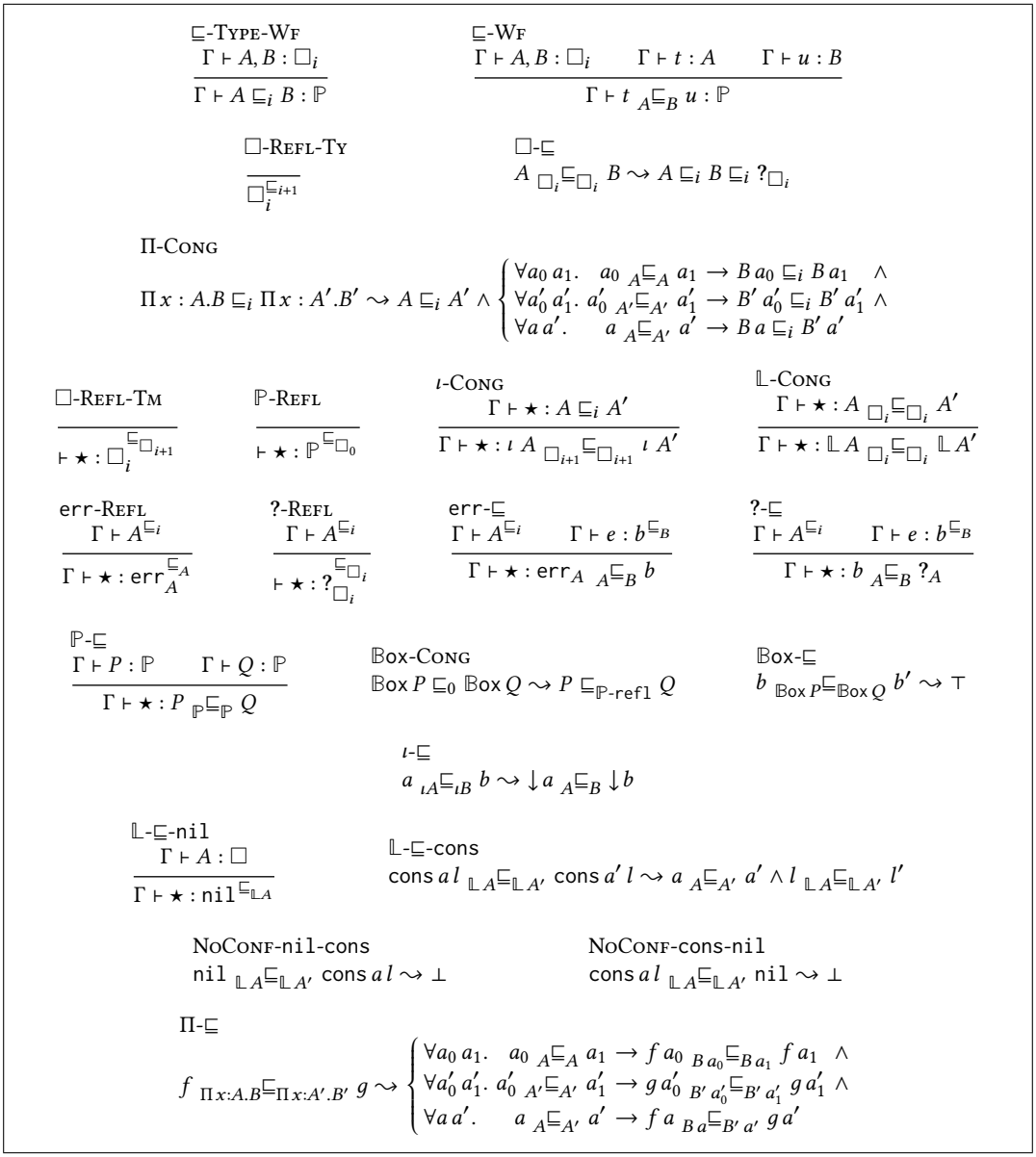

Fig. 4. Precision on types and terms

is the technical counterpart of the intuition given in §2.3 that precision between products should be guarded by an explicit use of cumulativity.

Next come the rules for type formers: all of them—apart, crucially, from product types— are either directly self-precise (as terms of $\square_i$) via □-Refl-Tm and ℙ-Refl[7] or congruent (ι-Cong, 𝕃-Cong). The two exceptional types $\mathtt{err}_\square$ and $?_\square$ are also self-precise, as a special case of err-Refl

[7]We use the wildcard ★ to avoid introducing extraneous names for inhabitants of strict propositions.

and ?-Refl. Via err-Refl, we also see that the previous reflexivity/congruence rules provide an exhaustive list of the types A such that $A \sqsubseteq_i ?_{\square_i}$, where, again, function types are missing.

As for terms, the generic rules err-Refl and ?-Refl ensure that $\mathtt{err}_A$ and $?_A$ are in relation with themselves, while err-⊑ and ?-⊑ say that they are respectively minimal and maximal—for self-precise terms of a self-precise type.

Heterogeneous precision between propositions is degenerate (ℙ-⊑), meaning that any two propositions are related by precision. Monotonicity of 𝔹ox with respect to precision on propositions (𝔹ox-Cong) means that precision between boxed propositions is degenerate as well. To validate this, we endow 𝔹ox types with a precision relation collapsing all terms (𝔹ox-⊑). This is sensible, as it showcases the fact that no (self-precise) context should be allowed to distinguish propositions, since those, even 𝔹oxed, ought to be observationally subsingletons. It also makes the eager erroring behavior of Box-Box sensible, since the error is as good an inhabitant of a 𝔹oxed proposition as any.

Cumulativity preserves the relation between types coming from lower levels (ι-⊑), meaning that coercions between a type and its lifting are monotone. On inductive types the precision relation closely resembles binary parametricity [Bernardy et al. 2012], relating a constructor to itself when arguments are related (𝕃-⊑-nil, 𝕃-⊑-cons). Outside of the diagonal, two no confusion principles (NoConf-nil-cons, NoConf-cons-nil) allow to deny the relatedness of lists that have distinct head constructors.[8]

Finally, we need to explain how function types are related by (type) precision. For simplicity, we start with the non-dependent case that takes the standard shape found in other gradual languages: two function types $A \to B$ and $A' \to B'$ are related whenever their domains and codomains are related: $A \sqsubseteq_i A' \wedge B \sqsubseteq_i B'$. The relation of precision $f \ {}_{A\to B}\sqsubseteq_{A'\to B'} \ g$ between functions $f : A \to B$ and $g : A' \to B'$ has to ensure that (1) f is monotone with respect to the precision on A and B; (2) g is monotone with respect to the precision on A' and B'; and (3) given inputs $a : A, a' : A'$ related by precision $a \ {}_A\sqsubseteq_A \ a'$, $f\,a : B$ is related to $g\,a' : B'$ by ${}_B\sqsubseteq_B$. Condition (3) boils down to the standard definition of (binary) parametricity on function types. Additional conditions (1-2) are required to ensure quasi-reflexivity at function types: since we do not want to globally impose that functions respect precision, we need to explicitly require that precision only relates monotone functions. Note that for a function $f : A \to B$ between self-precise types, being self-precise is logically equivalent to being monotone with respect to precision, meaning that the conditions (1-3) are equivalent in that case.

In the case of dependent function types (Π-Cong), domains must be related similarly to the non-dependent case but the codomains must now be related *as type families*, meaning that they are required to satisfy variants of the conditions (1-3) with respect to type precision. Finally, the relation between dependent functions is described by Π-⊑ and requires again that both functions are monotone and map related input to related outputs, at the adequate types.

Example 2 (Necessity of monotonicity in function types). Consider the two functions $\mathbb{0} \to \mathtt{unit}$ defined as $f := \mathtt{catch}_{\mathbb{0}}\ (\lambda(x : \mathbb{0}).\mathtt{unit})\ ()\ \mathtt{err}_{\mathbb{0}}$ and $g := \mathtt{catch}_{\mathbb{0}}\ (\lambda(x : \mathbb{0}).\mathtt{unit})\ ?_{\mathbb{0}}\ ()$. These functions verify that $\forall x \sqsubseteq_{\mathbb{0}} y, f\,x \sqsubseteq_{\mathtt{unit}} g\,y$, but neither f or g are monotone. As a consequence, precision on function types need to be restricted to monotone functions. Taking f to be instead the constant function with value $\mathtt{err}_{\mathtt{unit}}$, or g the constant function with value $?_{\mathtt{unit}}$ shows that we really need both functions to be monotone.

[8] In the case of lists, we can derive solely from these two rules that any non-exceptional constructor is discriminable from $\mathtt{err}_{\mathbb{L}A}$, $?_{\mathbb{L}A}$, *e.g.* that $\mathtt{nil}\ {}_{\mathbb{L}A}\not\sqsubseteq_{\mathbb{L}A}\ \mathtt{err}_{\mathbb{L}A}$, and $?_{\mathbb{L}A}\ {}_{\mathbb{L}A}\not\sqsubseteq_{\mathbb{L}A}\ \mathtt{err}_{\mathbb{L}A}$. For more general inductive types, these rules should be assumed primitively, *e.g.* $?_{\mathbb{0}}\ {}_{\mathbb{0}}\not\sqsubseteq_{\mathbb{0}}\ \mathtt{err}_{\mathbb{0}}$ for the empty type $\mathbb{0}$.

Quasi-reflexivity and transitivity

$$\lfloor - \rfloor : A \sqsubseteq_i B \to A \sqsubseteq_i A \qquad \text{Whenever } A^{\sqsubseteq_i}, B^{\sqsubseteq_i}, C^{\sqsubseteq_i}:$$

$$\lceil - \rceil : A \sqsubseteq_i B \to B \sqsubseteq_i B \qquad \lfloor - \rfloor : a\ {}_A\sqsubseteq_B\ b \to a\ {}_A\sqsubseteq_A\ a$$

$$- \cdot - : A \sqsubseteq_i B \to B \sqsubseteq_i C \to A \sqsubseteq_i C \qquad \lceil - \rceil : a\ {}_A\sqsubseteq_B\ b \to b\ {}_B\sqsubseteq_B\ b$$

$$- \cdot - : a\ {}_A\sqsubseteq_B\ b \to b\ {}_B\sqsubseteq_C\ c \to a\ {}_A\sqsubseteq_C\ c$$

Functoriality & Decomposition of casts

Cast-Id
$$\frac{A^{\sqsubseteq_i} \qquad a^{\sqsubseteq_A}}{\langle A \Leftarrow A\rangle\ a \mathrel{\rotatebox[origin=c]{180}{$\sqsubseteq$}}\!\!\sqsubseteq_A a}$$

Upcast-Comp
$$\frac{A \sqsubseteq_i B \qquad B \sqsubseteq_i C \qquad a^{\sqsubseteq_A}}{\langle C \Leftarrow B\rangle\ \langle B \Leftarrow A\rangle\ a \mathrel{\rotatebox[origin=c]{180}{$\sqsubseteq$}}\!\!\sqsubseteq_C \langle C \Leftarrow A\rangle\ a}$$

Downcast-Comp
$$\frac{A \sqsubseteq_i B \qquad B \sqsubseteq_i C \qquad c^{\sqsubseteq_C}}{\langle A \Leftarrow B\rangle\ \langle B \Leftarrow C\rangle\ c \mathrel{\rotatebox[origin=c]{180}{$\sqsubseteq$}}\!\!\sqsubseteq_A \langle A \Leftarrow C\rangle\ c}$$

Upper-decomposition
$$\frac{A \sqsubseteq_i X \qquad B \sqsubseteq_i X \qquad a^{\sqsubseteq_A}}{\langle B \Leftarrow X\rangle\ \langle X \Leftarrow A\rangle\ a \mathrel{\rotatebox[origin=c]{180}{$\sqsubseteq$}}\!\!\sqsubseteq_B \langle B \Leftarrow A\rangle\ a}$$

Characterization of heterogenous term precision

$$\text{For } A^{\sqsubseteq_i}, B^{\sqsubseteq_i}, \qquad a\ {}_A\sqsubseteq_B\ b \qquad \leftrightarrow \qquad a\ {}_A\sqsubseteq_A\ \langle B \Leftarrow A\rangle\ b \quad \wedge \quad b^{\sqsubseteq_B}$$

Fig. 5. Properties of precision

4.2 Properties of Precision

We now extend the theory with properties about precision that are validated by our model (presented in §5), in order to allow users to reason abstractly about precision proofs in GRIP. Thus, whenever we say that a property "holds" in this section, it should be understood as a twofold statement: first, the property is validated in the model, and so we add a new constant in GRIP, witnessing its truth.

Embedding-projection pairs. Why do we care so much about precision? The fundamental reason is that casts between types that are related by precision are well-behaved. We adopt the approach of New and Ahmed [2018] to characterize well-behaved pairs of casts as those that form an embedding projection pair (ep-pair). In our setting that allows non monotone functions, the definition of an ep-pair needs to be relativized to self-precise elements.

Definition 1 (Embedding projection pairs). *Let $A \sqsubseteq_i B$ be types related by precision, then the pair of casts $(\langle B \Leftarrow A\rangle, \langle A \Leftarrow B\rangle)$ satisfies the following:*

Monotonicity *both $\langle B \Leftarrow A\rangle$ and $\langle A \Leftarrow B\rangle$ are monotone with respect to precision,*

$$\forall a, a' : A, a\ {}_A\sqsubseteq_A\ a' \to \langle B \Leftarrow A\rangle\ a\ {}_B\sqsubseteq_B\ \langle B \Leftarrow A\rangle\ a'$$
$$\forall b, b' : B, b\ {}_B\sqsubseteq_B\ b' \to \langle A \Leftarrow B\rangle\ b\ {}_B\sqsubseteq_B\ \langle A \Leftarrow B\rangle\ b'$$

Adjunction *for any self-precise terms $a : A, b : B$ the following adjunction property is verified*

$$a^{\sqsubseteq_A} \wedge b^{\sqsubseteq_B} \quad \to \quad \langle B \Leftarrow A\rangle\ a\ {}_B\sqsubseteq_B\ b \iff a\ {}_A\sqsubseteq_B\ b \iff a\ {}_A\sqsubseteq_A\ \langle A \Leftarrow B\rangle\ b,$$

Retraction *a self-precise term $a : A$ is equiprecise (i.e., both more and less precise, noted $\mathrel{\rotatebox[origin=c]{180}{$\sqsubseteq$}}\!\!\sqsubseteq_A$) with its downcast-upcast:*

$$a^{\sqsubseteq_A} \quad \to \quad \langle A \Leftarrow B\rangle\ \langle B \Leftarrow A\rangle\ a \mathrel{\rotatebox[origin=c]{180}{$\sqsubseteq$}}\!\!\sqsubseteq_A a.$$

*We call $\langle B \Leftarrow A\rangle - : A \to B$ the **upcast** associated to the ep-pair and $\langle A \Leftarrow B\rangle - : B \to A$ the **downcast**.*

Order-like properties. In order to establish that two types are related by precision, we can the properties of the precision relations described in Fig. 5 beside those of Fig. 4. Type precision is a quasi-reflexive and transitive relation, and so is term precision at any self-precise type, meaning that ${}_A\sqsubseteq_A$ is quasi-reflexive and transitive whenever $A^{\sqsubseteq_i}$. Moreover, using Fig. 4, they admit $\mathtt{err}$ and $?$ as respectively smallest and largest (self-precise) elements. More generally, heterogeneous term precision satisfies indexed variants of quasi-reflexivity and transitivity on self-precise types. Combining these with the embedding-projection properties induced by precision-related types give rise to a rich "yoga" of indexed adjoints, deriving for instance properties like:

$$A \sqsubseteq_i B \sqsubseteq_i C \to a\ {}_A\sqsubseteq_C\ c \to \langle B \Leftarrow A\rangle\, a\ {}_B\sqsubseteq_C\ c$$

Composing casts. The ep-pair induced by precision are functorial: casting a self-precise term a of self-precise type A to A itself is equiprecise to a (Cast-Id), a succession of upcasts between precision-related types combine to a single upcast (Upcast-Comp) and similarly for downcasts (Downcast-Comp). A further fundamental property of casts is that they decompose through any type less precise than both the source and target of the cast: if $A \sqsubseteq_i X$ and $B \sqsubseteq_i X$, then for any self-precise term $a : A$, the cast $\langle B \Leftarrow A\rangle\, a$ is equiprecise to an upcast from A to X followed by a downcast to B:

$$a^{\sqsubseteq_A} \to \langle B \Leftarrow X\rangle\, \langle X \Leftarrow A\rangle\, a \rightleftharpoons_B \langle B \Leftarrow A\rangle\, a$$

In particular, whenever $A, B : \square_i$ are more precise than $?_{\square_i}$, that is when A, B are self-precise as terms of $\square_i$, $?_{\square_i}$ provides such a common upper bound for precision. In that favorable situation, casts from A to B can be analyzed through the ep-pairs with $?_{\square_i}$.

Heterogeneous precision. We can now explain how heterogenous term precision ${}_A\sqsubseteq_B$ can be alternatively characterized using only its restrictions to terms of the same type and casts. Let X be a type such that $A \sqsubseteq_i X$ and $B \sqsubseteq_i X$, then

$$\begin{aligned} a\ {}_A\sqsubseteq_B\ b \quad &\leftrightarrow \quad a\ {}_A\sqsubseteq_X\ \langle X \Leftarrow B\rangle\, b && (1)\\ &\leftrightarrow \quad \langle X \Leftarrow A\rangle\, a\ {}_X\sqsubseteq_X\ \langle X \Leftarrow B\rangle\, b && (2)\\ &\leftrightarrow \quad a\ {}_X\sqsubseteq_X\ \langle A \Leftarrow X\rangle\, \langle X \Leftarrow B\rangle\, b. && (2')\end{aligned}$$

The first equivalence (1) hold by transitivity with $b\ {}_B\sqsubseteq_X\ \langle X \Leftarrow B\rangle\, b$ for the left to right implication, and using the adjunction property of downcast as well as the retraction property $\langle B \Leftarrow X\rangle\, \langle X \Leftarrow B\rangle\, b\ {}_B\sqsubseteq_B\ b$ for the other implication. The equivalences (2) and (2') are respectively the adjunction property of the upcast and downcast associated to $A \sqsubseteq_i X$. Informally, (2) gives a simple explanation of heterogenous precision between terms as precision between the upcasts in any upper bound. Thanks to Upper-decomposition, (2') can be reformulated without any mention to the bound X, using solely precision on A and B:

$$a\ {}_A\sqsubseteq_B\ b \quad \leftrightarrow \quad a\ {}_A\sqsubseteq_A\ \langle B \Leftarrow A\rangle\, b \quad \wedge \quad b^{\sqsubseteq_B}$$

This last equivalence between heterogeneous and homogeneous term precision actually holds for any pair of self-precise types A, B.

Failure of threesomes. Since casts decompose in a well-behaved way through any upper bound, it is natural to wonder whether a similar property would hold for lower bounds, as can be found in threesomes [Siek and Wadler 2010] in the simply-typed gradual setting. In general, if $Y \sqsubseteq_i A$, $Y \sqsubseteq_i B$ we can derive from properties of casts that for any self-precise term $a : A$, $\langle B \Leftarrow Y\rangle\, \langle Y \Leftarrow A\rangle\, a\ {}_B\sqsubseteq_B\ \langle B \Leftarrow A\rangle\, a$, and taking $A = B = \mathbb{N}$, $Y = \mathtt{err}_{\square_i}$ and $a = 0$ shows that this precision ordering can be strict. We could still expect that this relation is an equiprecision when Y is sufficiently close to both A and B, typically when it is their meet $A \sqcap B$ for the precision relation. Such a condition is known

as the Beck-Chevalley condition in the literature on hyperdoctrines and descent [Lawvere 1970], and the following counterexample shows that this property does not hold in GRIP.

Example 3 (No cast decomposition through meets). Let $X_1 = \mathbb{N} \to \mathbb{N}$ and $X_2 = \Pi(b : \mathbb{B})(\text{if } x \text{ then } \mathbb{N} \text{ else } \mathbb{B})$. Computing their meet gives

$$\begin{aligned} X_1 \sqcap X_2 &= \Pi(x : \mathbb{N} \sqcap \mathbb{B})\ \mathbb{N} \sqcap (\text{if } \langle \mathbb{B} \Leftarrow \mathbb{N} \sqcap \mathbb{B} \rangle\ x \text{ then } \mathbb{N} \text{ else } \mathbb{B}) \\ &= \Pi(x : \mathtt{err}_\square)\ \mathbb{N} \sqcap (\text{if } \mathtt{err}_{\mathbb{B}} \text{ then } \mathbb{N} \text{ else } \mathbb{B}) \\ &= \Pi(x : \mathtt{err}_\square)\ \mathbb{N} \sqcap \mathtt{err}_\square \\ &= \mathtt{err}_\square \to \mathtt{err}_\square \end{aligned}$$

Now computing the result of casting $f : X_1 := \lambda(n : \mathbb{N}).5$ to X_2 directly and through $X_1 \sqcap X_2$, and evaluating both results on $\mathtt{true}$, we obtain

$$\begin{aligned} (\langle X_2 \Leftarrow X_1 \rangle\ f)\ \mathtt{true} &= (\lambda(b : \mathbb{B}).\langle \text{if } b \text{ then } \mathbb{N} \text{ else } \mathbb{N} \Leftarrow \mathbb{N} \rangle\ 5)\ \mathtt{true} \\ &= \langle \text{if } \mathtt{true} \text{ then } \mathbb{N} \text{ else } \mathbb{N} \Leftarrow \mathbb{N} \rangle\ 5 \\ &= \langle \mathbb{N} \Leftarrow \mathbb{N} \rangle\ 5 = 5 \end{aligned}$$

and

$$\begin{aligned} (\langle X_2 \Leftarrow X_1 \sqcap X_2 \Leftarrow X_1 \rangle\ f)\ \mathtt{true} &= (\langle X_2 \Leftarrow \mathtt{err}_\square \to \mathtt{err}_\square \Leftarrow X_1 \rangle\ f)\ \mathtt{true} \\ &= (\langle \mathtt{err}_\square \to \mathtt{err}_\square \Leftarrow X_1 \rangle\ \lambda(x : \mathtt{err}_\square).\, \mathtt{err}_{\mathtt{err}_\square})\ \mathtt{true} \\ &= (\lambda(b : \mathbb{B}).\, \mathtt{err}_{\text{if } b \text{ then } \mathbb{N} \text{ else } \mathbb{B}})\ \mathtt{true} \\ &= \mathtt{err}_{\mathbb{N}} \end{aligned}$$

In particular, $\langle X_2 \Leftarrow X_1 \rangle\ f \not\sqsubseteq \langle X_2 \Leftarrow X_1 \sqcap X_2 \Leftarrow X_1 \rangle\ f$ and the cast from X_1 to X_2 cannot be decomposed through a type more precise than both X_1 and X_2. This counterexample can be adapted to use dependent sums Σ instead of dependent products, showing that this phenomenon is proper to type dependency and function types are not crucial.

Note that all the properties presented in this section only apply to self-precise terms. The behavior of cast on types or terms that are not self-precise, typically non monotone functions, is left partially unconstrained.

4.3 Monotone Fragment

By adequately restricting GRIP, we can consider a fragment where every term is monotone. On that fragment precision between functions only needs a single heterogeneous component, bypassing boilerplate proofs of monotonicity. In practice, a characterization of this fragment could be used to automatically synthesize monotonicity proofs and lift a sizeable share of the burden imposed to the programmer. The quest for a large monotone fragment explains our design decision to degenerate precision on $\mathbb{P}$: doing so allows for a monotone negation on propositions.

There are two main non-monotone features in GRIP. One is obviously the `catch` constructor, which purposely allows for a non-monotone treatment of errors and ?. More insidious is the Π term constructor, for fundamental reasons linked with the Fire Triangle of Graduality. However, Lennon-Bertrand et al. [2022] show how to avoid this by systematically lifting product types one universe level up in their CIC$^\uparrow$ system—the only variant of CastCIC which satisfies both normalization and graduality by sacrificing conservativity over CIC—, which we can embed in GRIP. We can therefore rethink CastCIC$^\uparrow$ as an attempt to guarantee that every well-typed term is self-precise in order to globally satisfy graduality.

Monotone `catch`. The typical non-monotone construction in GRIP, is the `catch` construction on inductive types (see Example 2). However there is a generic way to proves that a `catch` is monotone, assuming adequate precision hypotheses on its arguments. In the case of lists, monotonicity of $\mathtt{catch}^{\square}_{\mathbb{L}\,A}$ amounts to:

$$\forall l\, l'.\ l\ {}_{\mathbb{L}\,A}\sqsubseteq_{\mathbb{L}\,A}\ l' \to \mathtt{catch}^{\square}_{\mathbb{L}\,A}\ P\ h_{\mathtt{nil}}\ h_{\mathtt{cons}}\ h_{\mathtt{err}}\ h_?\ l\ {}_{P\,l}\sqsubseteq_{P\,l'}\ \mathtt{catch}^{\square}_{\mathbb{L}\,A}\ P\ h_{\mathtt{nil}}\ h_{\mathtt{cons}}\ h_{\mathtt{err}}\ h_?\ l'$$

A natural proof proceeds by successive induction on l and l' using $\mathtt{catch}^{\mathbb{P}}$. The cases with distinct head constructors, e.g. $l = \mathtt{nil}, l' = \mathtt{cons}\ a\ l''$, are contradictory thanks to the no-confusion rules for precision on list (for instance NoConf-nil-cons). For the valid cases, we need to assume that the the branches $h_{\mathtt{nil}}$ and $h_{\mathtt{cons}}$ are less precise than $h_{\mathtt{err}}$ and more precise than $h_?$, and that $h_{\mathtt{cons}}$ is self-precise, e.g. $h_{\mathtt{err}}\ {}_{P\,\mathtt{err}_{\mathbb{L}\,A}}\sqsubseteq_{P\,\mathtt{nil}}\ h_{\mathtt{nil}}$.

Embedding CastCIC$^{\uparrow}$ *in* GRIP. We can achieve an embedding of CastCIC$^{\uparrow}$ into GRIP by using explicit cumulativity to systematically increase the universe level of Π types, mimicking the typing rule of products in the former system:

$$\frac{\Gamma \vdash_{\mathsf{CastCIC}^{\uparrow}} A : \square_i \qquad \Gamma, x : A \vdash_{\mathsf{CastCIC}^{\uparrow}} B : \square_i}{\Gamma \vdash_{\mathsf{CastCIC}^{\uparrow}} \Pi x : A.B : \square_{i+1}}\ \Pi\text{-CastCIC}^{\uparrow}$$

To distinguish the two, we use $\vdash_{\mathsf{GRIP}}$ for judgments in GRIP, and $\vdash_{\mathsf{CastCIC}^{\uparrow}}$ for judgments in CastCIC$^{\uparrow}$, that consist of the rules from Fig. 1, with the exception of rule Prod replaced by rule Π-CastCIC$^{\uparrow}$ above.[9]

It is rather straightforward to define a translation $[-]$ from CastCIC$^{\uparrow}$ to GRIP: the translation preserves all term and type constructor but Π types where it adds an explicit coercion of cumulativity:

$$\begin{array}{lcl} [\Pi\, x : A.B] & := & \iota\,(\Pi\, x : [A].[B]) \\ [\lambda\, x : A.t] & := & \uparrow(\lambda\, x : [A].[t]) \\ [t\ u] & := & (\downarrow [t])\ [u] \end{array}$$

Extending this translation to contexts in a pointwise fashion, we obtain the following correctness lemma.

Lemma 4. *The translation* $[-]$ *from* CastCIC$^{\uparrow}$ *to* GRIP *forms a syntactic model:*

(1) If $\Gamma \vdash_{\mathsf{CastCIC}^{\uparrow}} t : A$ *and* $t \rightsquigarrow t'$ *in* CastCIC$^{\uparrow}$ *then* $[\Gamma] \vdash_{\mathsf{GRIP}} [t] \equiv [t'] : [A]$ *in* CastCIC*;*

(2) If $\Gamma \vdash_{\mathsf{CastCIC}^{\uparrow}} t : A$ *then* $[\Gamma] \vdash_{\mathsf{GRIP}} [t] : [A]$

Proof. For point (1), β-reduction is preserved thanks to Coe-Retr and the reduction rules for casts can be simulated, in particular the decomposition of Π types into $?_{\square}$. Point (2) is then immediate from the observation that Π-CastCIC$^{\uparrow}$ can be translated to an application of Prod followed by Cum, Abs is translated to an application of the same rule followed by Coe, and App is modified with an application of Coe-Inv. □

Theorem 5 (Self-precision of CastCIC$^{\uparrow}$ embedding). *If* $\vdash_{\mathsf{CastCIC}^{\uparrow}} t : A$ *then* $\vdash_{\mathsf{GRIP}} \star : [t]^{\sqsubseteq_{[A]}}$.

Proof. We prove more generally that if $\Gamma \vdash_{\mathsf{CastCIC}^{\uparrow}} t : A$ then $[\Gamma]_{\varepsilon} \vdash_{\mathsf{GRIP}} \star : [t]_0\ {}_{[A]_0}\sqsubseteq_{[A]_1}\ [t]_1$, where $[\Gamma, x : A]_{\varepsilon} := [\Gamma]_{\varepsilon}, x_0 : [A]_0, x_1 : [A]_1, x_{\varepsilon} : x_0\ {}_{[A]_0}\sqsubseteq_{[A]_1}\ x_1$, and $[x]_i := x_i$. The proof proceeds by induction on the typing derivation, we only treat the central case $t = \Pi\, x : A.B$. By induction hypothesis, we have $[\Gamma]_{\varepsilon} \vdash_{\mathsf{GRIP}} ih_A : [A]_0\ {}_{[\square_i]_0}\sqsubseteq_{[\square_i]_1}\ [A]_1$ and $[\Gamma]_{\varepsilon}, x_0 : [A]_0, x_1 : [A]_1, x_{\varepsilon} : [A]_0\ {}_{[\square_i]_0}\sqsubseteq_{[\square_i]_1}\ [A]_1 \vdash_{\mathsf{GRIP}} ih_B : [B]_0\ {}_{[\square_i]_0}\sqsubseteq_{[\square_i]_1}\ [B]_1$.

[9] Lennon-Bertrand et al. [2022] have more permissive rule with respect to universe levels than we do, by baking in some cumulativity in the formation of Π-types and casts.

We need to prove that $\iota(\Pi\, x_0 : [A]_0.[B]_0)\ {}_{[\square_{i+1}]_0}\sqsubseteq_{[\square_{i+1}]_1}\ \iota(\Pi\, x_1 : [A]_1.[B]_1)$, hence using ι-Cong to $\Pi\, x_0 : [A]_0.[B]_0 \sqsubseteq_i \Pi\, x_1 : [A]_1.[B]_1$. The two heterogeneous precision required by Π-Cong are direct consequences of ih_A and ih_B using $\square$-$\sqsubseteq$ to relate type and term precision at level i. Finally, the monotonicity of $[B]_0$ and $[B]_1$ are consequences of ih_B and quasi-reflexivity of precision that holds because every type in the context is self-precise. □

5 A MODEL OF A REASONABLY GRADUAL TYPE THEORY

In this section we prove Theorem 1, that is the relative consistency of GRIP with a hierarchy of n universes with respect to MLTT[10] with $(n+1)$ universes and a type of definitionally proof irrelevant propositions. To do so, we exhibit a model where types are equipped with a relation reflecting precision. We formalized the components of this model (for two universes $\square_0$ and $\square_1$) in Coq. The construction of the model can be stratified in 3 layers:

- first, a computational layer that provides meaning to casts and exceptional terms $\mathtt{err}_A, ?_A$;
- second, a relational layer that equips every type with a relation and defines a compatible global heterogeneous relation between elements;
- third, a logical layer ensuring that said relations do capture well-behaved casts whenever all inputs are adequately related.

Computational layer. The computational layer closely resembles the discrete model of [Lennon-Bertrand et al. 2022], and we explain here its main features. The introduction of exceptional terms follows the approach of ExTT [Pédrot and Tabareau 2018]. Its main point is to extend each inductive type with two new constructors, one for **?** and one for `err`. Product types and functions are left unmodified, defining **?** and `err` pointwise.

We depart from this model on universes, so that we can define the cast primitive by case analysis on types. Taking inspiration from Boulier et al. [2017], we interpret types as *codes* when they are seen as terms, and as *the semantics of those codes* when they are seen as types. Thus, the standard interpretation for a term inhabiting a type is maintained, but a function taking as argument an element of the universe $\mathbb{\square}_i$ can now perform a case analysis on the code of the type. The precise construction of the interpretation of the universe hierarchy employs a technique presented in [Sattler and Vezzosi 2020]. We first define an inductive family $\mathsf{code} : \square_i \to \square_{i+1}$ describing codes for types and then pack it as $\mathbb{\square}_i : \square_{i+1} := \Sigma(A : \square_i)\ \mathsf{code}\ A$, using the first projection as decoding. We can then define an operation `cast : forall (A B : `$\mathbb{\square}_i$`), A → B` by induction on theses codes, following the reduction rules of Fig. 2. Fig. 6 presents a simplified version of this construction, to which codes for the translation of the types $\mathtt{err}_{\square_i}$, $?_{\square_i}$, $\mathbb{P}$, $\mathbb{L}\, A$ and $\square_j$ (for $j < i$) are added in the actual development.

The exceptional model of Pédrot and Tabareau [2018] leave the interpretation of exceptions at the universe $\square$ unspecified. We exploit this underspecification, and define $\mathtt{err}_{\square_i}$ as the unit type $\mathbb{1}$ with a single element. $?_{\square_{i+1}}$ is interpreted by an inductive type `unknown` (Fig. 7) closed by all type constructors but dependent functions. Beyond the two constructors `err_unknown` and `unk_unknown` interpreting respectively $\mathtt{err}_{?_\square}$ and $?_{?_\square}$, `univ_unknown` allows to embed the preceding universe, `cum_unknown` hosts any type from said preceding universe (including product types), and `list_unknown` can be used to embed lists of elements from $\mathbb{\square}_{i+1}$. Additional inductive types would be represented with supplementary constructors. The interpretation of $?_{\square_0}$ do not use `univ_unknown` and `cum_unknown`.

[10] With the standard type formers $\mathbb{0}$, $\mathbb{1}$, $\mathbb{2}$, W, Σ, Id and Π.

```
Let El X : □ := X.1.

Inductive code : □_i → □_{i+1} :=
| code_Nat : code Nat
| code_Pi (A : □_i) (B : El A → □_i) :
  code (forall a, El (B a))
| ...
where □_i := (Σ(A : □_i) code A).
```

```
Fixpoint cast (A B : □_i) : A → B :=
  match A.2, B.2 with
  | code_Nat, code_Nat ⇒ λ n ⇒ n
  | code_Nat, code_Pi _ _ ⇒ λ _ ⇒ err _
  | code_Pi A0 A1, code_Nat ⇒ λ _ ⇒ err _
  | code_Pi A0 A1, code_Pi B0 B1 ⇒
    λ (f : forall a, El (A1 a)) (b : El B0) ⇒
    cast (A1 _) (B1 b) (f (cast B0 A0 b))
  | ... .
```

Fig. 6. Simplified code for the universe of codes and of cast.

```
Inductive unknown :=
| err_unknown
| univ_unknown (A : □_i)
| cum_unknown (A : □_i) (a : El A)
| list_unknown (l : list unknown)
| unk_unknown.
```

```
Inductive prec_unk : unknown → unknown → SProp :=
| err_any : sp x → prec_unk err_unknown x
| unk_any : sp x → prec_unk x unk_unknown
| univ_prec : A □_i⊑□_i B →
  prec_unk (univ_unknown A) (univ_unknown B)
| cum_prec A a B b : sp A → sp B → hprec A B a b →
  prec_unk (cum_unknown A a) (cum_unknown B b)
| list_prec l1 l2 : lift_list prec_unk l1 l2 →
  prec_unk (list_unknown l1) (list_unknown l2).
```

Fig. 7. Translation of $?_{\square}$ and its precision relation.

Relational layer. We now endow the translation of every type with a homogeneous relation `prec : forall (A : □_i), A → A → SProp`. Thanks to the characterization of heterogeneous precision in Fig. 5, we can use `prec` together with `cast` to obtain an heterogeneous relation on all types at the same universe level:

```
Let hprec (A B : □_i) (a : El A) (b : El B) : SProp := prec A a (cast A B b) ∧ prec B b b.
```

The construction of `prec` proceeds first by induction on the universe level, and then by induction on the code of the type. The cases for $\mathtt{err}_{\square_i}$, $\mathbb{P}$, inductive types, dependent functions and cumulativity injection follow the formulae given for precision in Fig. 4. In particular, defining homogeneous precision at function types relies on heterogeneous precision on the codomain. On universes, we use precision for the smaller universe, obtained by induction hypothesis on the universe level. The precision for unknown is described on the right of Fig. 7. `err_unknown` and `unk_unknown` are respectively smaller and larger than self-precise terms of any summand. `univ_prec` embeds the relation from $\square_i$ and `cum_prec` relate elements of self-precise types using the heterogeneous relation determined by $\square_i$. Finally, `list_prec` lifts the precision on unknown to lists.

Property layer. Once all definitions are in place, we need to show that the relations thus defined do characterize well-behaved casts. This is summarized by the following definitions.

Definition 2 (Partial preorder). *A partial preorder on a type X is a transitive and quasi-reflexive relation.*

A pair of functions $f : X \to Y$, $g : Y \to X$ between ppos X, Y forms an *embedding projection pair* if it satifisfies the condition of Definition 1. A *type family with casts* consists of a type family $B : A \to \square$ equipped with two functions $\Uparrow^B_{a,a'} : B\,a \to B\,a'$ and $\Downarrow^B_{a,a'} : B\,a' \to B\,a$.

Definition 3 (Indexed partial preorder). *If A is a partial preorder and B a type family with cast such that each $B\,a$ is endowed with a relation ${}_{B\,a}\sqsubseteq_{B\,a}$, then B is an indexed partial preorder when*

- *whenever $a^{\sqsubseteq_A}$, ${}_{B\,a}\sqsubseteq_{B\,a}$ is a partial preorder;*
- *if $a\ {}_A\sqsubseteq_A\ a'$, then $(\Uparrow^B_{a,a'}, \Downarrow^B_{a,a'})$ forms an ep-pair;*
- *whenever $a^{\sqsubseteq_A}$, $b^{\sqsubseteq_{B\,a}}$, $\Uparrow^B_{a,a}\ b \sqsupseteq_{B\,a}\ b \sqsupseteq_{B\,a} \Downarrow^B_{a,a}\ b$;*
- *if $a_0\ {}_A\sqsubseteq_A\ a_1\ {}_A\sqsubseteq_A\ a_2$, $b^{\sqsubseteq_{B\,a_0}}$, then $\Uparrow^B_{a_1,a_2}\Uparrow^B_{a_0,a_1}\ b \sqsupseteq_{B\,a_2}\Uparrow^B_{a_0,a_2}\ b$ and $\Downarrow^B_{a_2,a_1}\Downarrow^B_{a_1,a_0}\ b \sqsupseteq_{B\,a_2}\Downarrow^B_{a_2,a_0}\ b$*

Now the model validates the following:

Theorem 6 (Properties of precision). *The universes $\mathbb{\square}_i$ is a partial preorder for term and type precision and the type families $\mathrm{El} : \mathbb{\square}_i \to \square_i$ equiped with* `cast` *are indexed partial preorders.* `unknown` *is a greatest element for term precision on the universe.*

The proof of this theorem proceed by induction on multiset of codes, showing that the relation induced by a code is partial preorder, that pairs of casts between the partial preorders induced by a pair of codes form an ep-pair and that the eppairs induced by a triple of code compose adequately. To that end, we prove and use a lemmas asserting that type constructors from $\mathbb{\square}_i$ preserve partial preorders and ep-pairs, e.g. the relation on $\forall a : \mathrm{El}\,A, \mathrm{El}(B\,a)$ induced by a code `code_Pi A B` is a partial preorder whenever `A` is a partial preorder and B is an indexed partial preorder.

The properties presented in §4.2 are consequences of this theorem, using crucially the decomposition of heterogeneous term relation through any upper bound for the precision relation, the fact that any type at universe level i is bounded by $?_{\square_{i+1}}$ and cumulativity preserves and reflects precision.

6 EXTENSIONS AND IMPLEMENTION OF GRIP

We now discuss several extensions of GRIP for future work, and the provided Agda implementation.

6.1 Observational Equality

GRIP features two kinds of sorts, $\square$ for (impure) computationally relevant types and $\mathbb{P}$ for definitionally proof irrelevant propositions. The main purpose of $\mathbb{P}$ is to be able to define precision internally in GRIP, by induction on types. In the recent work of Pujet and Tabareau [2022], $\mathbb{P}$ is used in the same way to define a notion of observational equality by induction on types, satisfying extensionality principles. It turns out that internal precision and observational equality can both be integrated in GRIP, the transport primitive (defined in Transport) being the *safe* version of casts, where a proof of equality between the two types ensures the absence of cast errors.

There are two main interests in adding a notion of observational equality to GRIP. First, it allows us to state many properties than cannot be only stated using internal precision. For instance, equality is necessary to express internally what antisymmetry means for internal precision, and prove that it holds on types for which all terms are self-precise. Second, it provides a canonical way to express (non-gradual) subset types in GRIP, thus recovering a flavour of indexed inductive types.

6.2 Indexed Inductive Types in $\mathbb{P}$

It is possible to add general index inductive types in $\mathbb{P}$, such as less or equal in $\mathbb{N}$. Gilbert et al. [2019] describe a general criterion to detect which inductive types in $\mathbb{P}$ can be eliminated into $\square$. Basically this criterion amounts to detect when an indexed inductive type can be encoded with a fixpoint over the indices. This criterion also works for GRIP, and could be reused directly.

6.3 From GRIP to a Gradual Proof Assistant

GRIP is still quite far from a real-life proof assistant. As explained at the beginning of §2, usual gradual systems are separated into two languages: a source language where types are compared in an optimistic way using the wildcard **?**, and a target language with casts to explicitely flag where those optimistic assumptions are made, so as to be able to raise errors in case of type incompatibilities discovered during program evaluation. Here we concentrated on designing the target language, as our contributions apply mostly to it, with the pragma that the source and elaboration layers as presented in [Lennon-Bertrand et al. 2022] could be easily adapted to our extensions. Consequently, we chose to present our type theory in a standard, undirected fashion, rather than using the bidirectional approach of Lennon-Bertrand et al. [2022][11]. However, building an actual proof assistant involves tackling that elaboration layer, and the many subtle points it involves, which were only partially solved by [Lennon-Bertrand et al. 2022]. One example would be the interaction between unification (the main and crucial feature of elaboration in *e.g.* Coq) and gradual features of the language, especially consistency.

But even if one considers only the target language, incorporating it in an actual proof assistant is no small feat. In GRIP, we made a wealth of technical choices (impredicativity of $\mathbb{P}$, explicit cumulativity, and so on) that might need to be reconsidered if one wishes to integrate gradual features in a proof assistant that takes a different path. In particular, a proper treatment of universe levels is a challenge. For instance, a system more flexible (and probably easier to use) than GRIP would allow casts between types at different levels, but this would cause an unprecedented dependency between reduction (of casts) and universe levels, which in turn raises subtle implementation questions.

Similarly, we made some choices in the definition of precision, both in the rules of Fig. 4 and the properties reflected in GRIP in §4.2. They were in part guided by the aim to make the system as ready for use as possible, but they might need to be reconsidered in a practical implementation.

Finally, an interesting design point pertains to the `catch` primitive. Actual proof assistants usually do not rely on recursors, but instead provide facilities for pattern-matching in various forms. Implementations of `catch` should be adapted to those. In particular, a mechanism to present monotone `catch` as presented in §4.3 could take inspiration from the implementation of higher inductive types, with path-constructors replaced by monotonicity constraints.

6.4 Proof-of-Concept in Agda

The Agda proof assistant is an extension of MLTT with many features, including a predicative sort of proof-irrelevant propositions as described in [Gilbert et al. 2019]. Additionally, recent versions of Agda feature rewrite rules [Cockx et al. 2020], a canonical way to postulate new primitives in the theory and reduction rules for them. This makes Agda a nice playground for prototype implementations of type theories. For instance, there is already a prototype implementation of exceptional type theory [Cockx et al. 2020] and of observational type theory [Pujet and Tabareau 2022]. We build on those to construct a toy implementation of GRIP, which is successfully accepted by the confluence checker bundled with the rewrite rules mechanism.

The main discrepancy between Agda and GRIP is that the sort Prop of Agda is predicative, which means that the definition of internal precision on types and terms needs to be slightly amended, in the following way:

$$\begin{array}{lcl} \mathsf{postulate}\ \_\sqsubseteq\_ & : & \mathsf{Set}_\ell \to \mathsf{Set}_\ell \to \mathsf{Prop}_{\ell+1} \\ \mathsf{postulate}\ \_\preccurlyeq\_ & : & \{A\ B : \mathsf{Set}_\ell\} \to A \to B \to \mathsf{Prop}_\ell \end{array}$$

[11] Such a presentation is recoverable if need be.

Note the presence of $\ell + 1$ in the level of the return type for type precision; this is necessary to make the reduction rule from term precision at the universe to type precision typecheck.

Reduction rules in GRIP are then encoded with a postulated equality and a companion rewrite rule. For instance, rule $\mathbb{L}$-Cong of Fig. 4 can be encoded as

```
postulate ⊑-Type-List : {A A′ : Set_ℓ} → List A ⊑ List A′ ≡ A ⊑ A′
{-# REWRITE ⊑-Type-List #-}
```

The complete implementation can be found in file grip-poc.agda, which relies on ett-rr.agda for implementation of exceptions, and setoid-rr.agda for implementation of observational type theory.

The examples exposed in §2 have been formally treated in grip-poc-example.agda. For instance, the formal proof of $\mathtt{mult}_{\mathbb{L}}^{\sqsubseteq}$ relies on the following lemma on $\mathtt{mult}_{\mathbb{L}}^{\mathtt{err}}$, characterizing its outputs on two lists that are in the precision relation, given the fact that the more precise one is not an error.

```
mult-err-mon : (l l′ : List Nat) → noterr-List l → l ≼ l′ →
               (mult^err_𝕃 l ≼ err_Nat ∧ (mult^err_𝕃 l′ ≼ err_Nat ∨ ?_Nat ≼ mult^err_𝕃 l′)) ∨
               (noterr-Nat (mult^err_𝕃 l) ∧ mult^err_𝕃 l ≼ mult^err_𝕃 l′)
```

As the reader can see, this lemma is far from a monotonicity statement, but it describes faithfully in which case $\mathtt{mult}_{\mathbb{L}}^{\mathtt{err}}$ raises an error, which is crucial to show that the function $\mathtt{mult}_{\mathbb{L}}$ is monotonous.

7 RELATED WORK

Effects in dependent type theory. Incorporating effects in type theory, specifically errors as needed for gradual systems, is particularly challenging. Indeed, the presence of effects triggers a strong tension with the metatheoretic properties of CIC, putting logical consistency in danger, as clarified by the Fire Triangle of Pédrot and Tabareau [2020]. Several programming languages mix dependent types with effectful computation, either giving up on metatheoretical properties, such as Dependent Haskell [Eisenberg 2016], which allows diverging type-level expressions, or by restricting the dependent fragment to pure expressions [Swamy et al. 2016; Xi and Pfenning 1998]. However, all hope is not lost, and [Pédrot and Tabareau 2017, 2018; Pédrot et al. 2019] build up from general considerations on effects to specifically consider exceptions, and end up introducing RETT, with the idea of having a separation between an effectful, inconsistent layer and a pure, consistent one to reason about the effectful one, which we take inspiration from in the design of GRIP.

Strict propositions and observational equality. It has long been recognized that equality in standard MLTT is too syntactic. Observational type theory [Altenkirch et al. 2007] was proposed to address this issue, but only thanks to work on incorporating (definitional) irrelevance in dependent type theory [Abel and Scherer 2012; Gilbert et al. 2019] was it possible to recently turn this proposition into a concrete system [Pujet and Tabareau 2022], by using the definitionally proof-irrelevant sort to host the observational equality. The sort $\mathbb{P}$ and the precision relation of GRIP are very much inspired respectively by the sort of definitionally proof-irrelevant propositions of Gilbert et al. [2019] and the observational equality of Pujet and Tabareau [2022].

Gradual typing and dependent types. This work continues a line of research in combining dependent types and dynamic type checking, as first explored by [Ou et al. 2004], more specifically following the gradual typing approach [Siek and Taha 2006; Siek et al. 2015], and extending it to a full-blown dependent type theory. Ou et al. [2004] study a programming language with separate dependently- and simply-typed fragments, using arbitrary runtime checks at the boundary. The blame calculus of Wadler and Findler [2009] considers subset types on base types, where the refinement is an arbitrary term, as in hybrid type checking [Knowles and Flanagan 2010], but lacks dependent function types. Tanter and Tabareau [2015] provide casts for subset types with decidable properties in Coq, and Dagand et al. [2018] support dependent interoperability [Osera et al. 2012] in Coq. All these approaches lack the notion of precision that is central to gradual typing. Gradual

refinement types [Lehmann and Tanter 2017] differ from the gradual subset types presented here in that they are an extension of liquid types [Rondon et al. 2008] with imprecise logical formulas, based on an SMT-decidable logic about base types. Eremondi et al. [2019] study the gradualization of CC_ω, and propose approximate normalization to ensure decidable typechecking. Approximate normalization satisfies the dynamic gradual guarantee, but not graduality in the sense of [New and Ahmed 2018], because casting to an imprecise type and back can yield the unknown term instead of the original term. The most recent and complete attempt to gradualize CIC, upon which we build in this work, is the study of GCIC and its underlying cast calculus CastCIC [Lennon-Bertrand et al. 2022], which comes under three variants. GRIP is an extension of CastCIC that allows for sound reasoning about gradual programs and, thanks to internal precision, can account for the specific form of graduality supported by $\mathsf{CastCIC}^{\mathcal{N}}$, the normalizing conservative extension of CIC, and can embed $\mathsf{CastCIC}^{\uparrow}$ as a subclass of terms that are self-precise.

REFERENCES

Andreas Abel and Gabriel Scherer. 2012. On Irrelevance and Algorithmic Equality in Predicative Type Theory. *Logical Methods in Computer Science* Volume 8, Issue 1 (3 2012). https://doi.org/10.2168/LMCS-8(1:29)2012

Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. 2007. Observational equality, now!. In *Proceedings of the Workshop on Programming Languages meets Program Verification (PLPV 2007)*. 57–68.

Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. 2012. Proofs for free: Parametricity for dependent types. *Journal of Functional Programming* 22, 2 (March 2012), 107–152.

Rastislav Bodík and Rupak Majumdar (Eds.). 2016. *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*. ACM Press, St Petersburg, FL, USA.

Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The next 700 syntactical models of type theory. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*. 182–194. https://doi.org/10.1145/3018610.3018620

Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. 2020. The Taming of the Rew: A Type Theory with Computational Assumptions. *Proceedings of the ACM on Programming Languages* (2020). https://hal.archives-ouvertes.fr/hal-02901011

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2015. Cubical Type Theory: a constructive interpretation of the univalence axiom. (May 2015), 262 pages.

Pierre-Évariste Dagand, Nicolas Tabareau, and Éric Tanter. 2018. Foundations of Dependent Interoperability. *Journal of Functional Programming* 28 (2018), 9:1–9:44.

Richard A. Eisenberg. 2016. Dependent Types in Haskell: Theory and Practice. arXiv:1610.07978 [cs.PL]

Joseph Eremondi, Éric Tanter, and Ronald Garcia. 2019. Approximate Normalization for Gradual Dependent Types. See[ICFP 2019 2019], 88:1–88:30.

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing, See [Bodík and Majumdar 2016], 429–442. See erratum: https://www.cs.ubc.ca/ rxg/agt-erratum.pdf.

Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. 2019. Definitional Proof-Irrelevance without K. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 1–28. https://doi.org/10.1145/3290316

ICFP 2019 2019.

Kenneth Knowles and Cormac Flanagan. 2010. Hybrid type checking. *ACM Transactions on Programming Languages and Systems* 32, 2 (Jan. 2010), Article n.6.

Bill Lawvere. 1970. Equality in hyperdoctrines and comprehension schema as an adjoint functor. In *Proceedings of the AMS Symposium on Pure Mathematics XVII*. 1–14.

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

Meven Lennon-Bertrand, Kenji Maillard, Nicolas Tabareau, and Éric Tanter. 2022. Gradualizing the Calculus of Inductive Constructions. *ACM Transactions on Programming Languages and Systems* 44, 2 (June 2022).

Per Martin-Löf. 1971. An Intuitionistic Theory of Types. Unpublished manuscript.

Max S. New and Amal Ahmed. 2018. Graduality from Embedding-Projection Pairs. , 73:1–73:30 pages.

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory.* Ph.D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

Ulf Norell. 2009. Dependently Typed Programming in Agda. In *Advanced Functional Programming (AFP 2008) (Lecture Notes in Computer Science, Vol. 5832)*. Springer-Verlag, 230–266.

Peter-Michael Osera, Vilhelm Sjöberg, and Steve Zdancewic. 2012. Dependent Interoperability. In *Proceedings of the 6th workshop on Programming Languages Meets Program Verification (PLPV 2012)*. ACM Press, 3–14.

Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. 2004. Dynamic Typing with Dependent Types. In *Proceedings of the IFIP International Conference on Theoretical Computer Science*. 437–450.

Pierre-Marie Pédrot and Nicolas Tabareau. 2017. An effectful way to eliminate addiction to dependence. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, 1–12. https://doi.org/10.1109/LICS.2017.8005113

Pierre-Marie Pédrot and Nicolas Tabareau. 2018. Failure is Not an Option - An Exceptional Type Theory. In *Proceedings of the 27th European Symposium on Programming Languages and Systems (ESOP 2018) (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer-Verlag, Thessaloniki, Greece, 245–271.

Pierre-Marie Pédrot and Nicolas Tabareau. 2020. The fire triangle: how to mix substitution, dependent elimination, and effects. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 58:1–58:28.

Pierre-Marie Pédrot, Nicolas Tabareau, Hans Fehrmann, and Éric Tanter. 2019. A Reasonably Exceptional Type Theory. See[ICFP 2019 2019], 108:1–108:29.

Loïc Pujet and Nicolas Tabareau. 2022. Observational Equality: Now For Good. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022). https://doi.org/10.1145/3498693

Emily Riehl and Michael Shulman. [n.d.]. A type theory for synthetic ∞-categories. ([n. d.]), 78.

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008)*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM Press, 159–169.

Christian Sattler and Andrea Vezzosi. 2020. Partial Univalence in n-truncated Type Theory. In *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller (Eds.). ACM Press, 807–819. https://doi.org/10.1145/3373718.3394759

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

Jeremy 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 (POPL 2010)*. ACM Press, Madrid, Spain, 365–376.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 32)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Asilomar, California, USA, 274–293.

Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent types and multi-effects in F*, See [Bodík and Majumdar 2016], 256–270.

Éric Tanter and Nicolas Tabareau. 2015. Gradual Certified Programming in Coq. In *Proceedings of the 11th ACM Dynamic Languages Symposium (DLS 2015)*. ACM Press, Pittsburgh, PA, USA, 26–40.

The Coq Development Team. 2020. *The Coq proof assistant reference manual*. https://coq.inria.fr/refman/ Version 8.12.

Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *Proc. ACM Program. Lang.* 3, ICFP, Article 87 (July 2019), 29 pages. https://doi.org/10.1145/3341691

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 (ESOP 2009) (Lecture Notes in Computer Science, Vol. 5502)*, Giuseppe Castagna (Ed.). Springer-Verlag, York, UK, 1–16.

Matthew Z Weaver and Daniel R Licata. 2020. A Constructive Model of Directed Univalence in Bicubical Sets. (2020), 14.

Hongwei Xi and Frank Pfenning. 1998. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '98)*. ACM Press, 249–257.