Gradual Indexed Inductive Types

MARA MALEWSKI, University of Chile, Chile
KENJI MAILLARD, Inria, France
NICOLAS TABAREAU, Inria, France
ÉRIC TANTER, University of Chile, Chile

Indexed inductive types are essential in dependently-typed programming languages, enabling precise and expressive specifications of data structures and properties. Recognizing that programming and proving with dependent types could benefit from the smooth integration of static and dynamic checking that gradual typing offers, recent efforts have studied gradual dependent types. Gradualizing indexed inductive types however remains mostly unexplored: the standard encodings of indexed inductive types in intensional type theory, e.g., using type-level fixpoints or subset types, break in the presence of gradual features; and previous work on gradual dependent types focus on very specific instances of indexed inductive types.

This paper contributes a general framework, named Punk, specifically designed for exploring the design space of gradual indexed inductive types. Punk is a versatile framework, enabling the exploration of the space between eager and lazy cast reduction semantics that arise from the interaction between casts and the inductive eliminator, allowing them to coexist and interoperate in a single system.

Our work provides significant insights into the intersection of dependent types and gradual typing, by proposing a criteria for well-behaved gradual indexed inductive types, systematically addressing the outlined challenges of integrating these types. The contributions of this paper are a step forward in the quest for making gradual theorem proving and gradual dependently-typed programming a reality.

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

Additional Key Words and Phrases: Gradual typing, proof assistants, dependent types

ACM Reference Format:
Mara Malewski, Kenji Maillard, Nicolas Tabareau, and Éric Tanter. 2024. Gradual Indexed Inductive Types. *Proc. ACM Program. Lang.* 8, ICFP, Article 255 (August 2024), 29 pages. https://doi.org/10.1145/3674644

1 Introduction

Indexed inductive types play a pivotal role in dependently-typed programming languages due to their versatility in expressing complex data structures and logical properties [Chlipala 2013; Pierce et al. 2015] and are therefore a predominant feature of languages such as Idris [Brady 2013] and proof assistants such as Coq [The Coq Development Team 2020], Lean [de Moura et al. 2015], and Agda [Norell 2007]. These types extend the capabilities of traditional inductive types by allowing constructor types to depend on terms, enabling precise and expressive specifications of data and program behavior. Some canonical examples of indexed inductive types include bounded natural numbers, length-indexed lists (called vectors), heterogeneous lists, Martin-Löf's propositional

*This work is partially funded by ANID/Doctorado Nacional/2021-21210982 and Inria Équipe Associée GRAPA.

Authors' Contact Information: Mara Malewski, University of Chile, PLEIAD Lab, Computer Science Department (DCC), Santiago, Chile, smalewsk@dcc.uchile.cl; Kenji Maillard, Inria, Gallinette Project-Team, Nantes, France, kenji.maillard@inria.fr; Nicolas Tabareau, Inria, Gallinette Project-Team, Nantes, France, nicolas.tabareau@inria.fr; Éric Tanter, University of Chile, PLEIAD Lab, Computer Science Department (DCC), Santiago, Chile, etanter@dcc.uchile.cl.

equality [Martin-Löf 1971], and lambda calculus terms indexed over names in scope. This expressive power makes dependently-typed languages invaluable not only for theorem proving but also for general-purpose programming: programmers can leverage indexed inductive types to encode intricate invariants and constraints within the type system, leading to safer and more reliable code. As programming languages, dependently-typed languages empower developers to write certified software by either including correctness proofs along the code (extrinsic style) or by extending data structures with invariants and using the type system to enforce correctness (intrinsic style); the latter approach uses indexed inductive types extensively.

While dependently-typed languages offer immense power, they are often quite challenging to use. The intricate type system and the need to express program properties at the type level creates a steep learning curve for both developers entering the realm of formal verification and mathematicians looking to mechanize their work.

Gradual typing is a discipline that provides flexibility by enabling the smooth transition between statically and dynamically typed code and vice versa [Siek and Taha 2006]. Code can be made more dynamic by using imprecise types, i.e., the unknown type **?** or types with unknown components, such as $? \to \mathbb{N}$. Conversely, code can be made more static by replacing occurrences of the unknown type with static types. Imprecision is checked optimistically by the typechecker, and type safety is ensured by inserting runtime checks—usually called *casts*—to detect any type error that may arise during execution. The smooth transition provided by gradual typing is captured by a couple of properties called the *gradual guarantees*, which state that typing and reduction are monotone with respect to precision [Siek et al. 2015b]. Gradual typing has been successfully applied to a wide range of type systems and programming language features. The impact of gradual typing is not limited to academia, as its ideas are being adopted by various mainstream programming languages, such as TypeScript, Flow, and Racket, among others.

There have been several attempts to integrate some degree of dynamic typing into dependently-typed languages [Dagand et al. 2018; Ou et al. 2004; Tanter and Tabareau 2015]. One remarkable effort to bring gradual typing to dependently-typed languages is the gradual dependently-typed language GDTL [Eremondi et al. 2019]. GDTL uses the Abstracting Gradual Typing methodology [Garcia et al. 2016] to gradualize the predicative variant of the calculus of constructions (CC_ω) with call-by-value semantics. GDTL has two distinct phases, a compile-time phase that optimistically approximates type checking (*approximate normalization*) and a runtime phase that, while being exact, may fail or diverge. Since GDTL allows for non-termination it is more suited for dependently-typed programming than for theorem proving.

Another notable effort is the Gradual Calculus of Inductive Constructions (GCIC) [Lennon-Bertrand et al. 2022]. GCIC brings gradual typing to a restricted version of Calculus of Inductive Constructions (CIC) that supports parametrized inductive types, not indexed inductive types in general. Lennon-Bertrand et al. [2022] have established that it is impossible for any gradual language to achieve a conservative extension of CIC, strong normalization, and graduality simultaneously. Graduality [New and Ahmed 2018], in this context, refers to a semantic formulation of the dynamic gradual guarantee where losing and then recovering precision yields an equivalent term. GCIC comes in three different variants, each one dropping one of these properties while maintaining the other two: $\mathrm{GCIC}^{\mathcal{N}}$ drops graduality, $\mathrm{GCIC}^{\mathcal{G}}$ drops strong normalization, and $\mathrm{GCIC}^{\uparrow}$ drops the conservative extension with respect to CIC.

GCIC has been recently extended in two separate directions. On the one hand, Eremondi et al. [2022] extend $\mathrm{GCIC}^{\mathcal{G}}$ with support for propositional equality. Like GDTL, their intention is to make a gradual dependently-typed language rather than a gradual theorem prover, hence they choose to sacrifice strong normalization. On the other hand, GRIP [Maillard et al. 2022] explores the space of gradual theorem proving by extending GCIC$^{\mathcal{N}}$ with an internalized notion of precision. GRIP features two universe hierarchies, one for gradual types and terms, whose reduction can raise an exception (either error or the unknown term), and another for pure (non-gradual) strict propositions [Gilbert et al. 2019]. Since the internalized notion of precision lives in the propositional universe, it is possible to reason in a *logically consistent* way about gradual programs. Although graduality does not hold universally in GRIP, the authors identify a subset of the language, called the *monotone fragment*, for which graduality holds and can be internally established. Outside of the monotone fragment are the Π types that produce terms of a universe and, more relevant to this work, the `catch` operator that serves as the eliminator of inductive types in an exceptional type theory [Pédrot and Tabareau 2018].

Despite the many advances in gradual type theories summarized above, none of these theories supports gradual indexed inductive types in general. In the gradual setting, handling indexed inductive types presents distinct challenges. On the one hand, traditional methods for encoding these types, such as type-level fixpoints and subset types, do not behave as expected when interacting with the unknown type and casts. On the other hand, direct approaches have either focused in a specific indexed inductive type [Eremondi et al. 2022] or ad-hoc mechanisms restricted to indices with decidable equality [Lennon-Bertrand et al. 2022]. This lack of support is a serious limitation considering the importance of indexed inductive types in all kinds of development with proof assistants such as Coq, Lean and Agda. This work tackles this challenge by extending GRIP with a general notion of gradual indexed inductive types. Interestingly, we observe that there is no unique or generally better way to deal with casts on indices at runtime: there are just different possible semantics for cast reduction, with different tradeoffs, just like there are different possible semantics for higher-order casts in standard gradually-typed languages [Siek et al. 2009].

Among the many behaviors one may consider for multiple consecutive casts, we concentrate on three options in this paper: free, meet, and forgetful. These approaches correspond to cast reduction semantics already studied in the gradual typing literature [Greenberg 2017; Siek and Wadler 2010] and exhibit different behaviors with respect to the interaction of cast sequences and elimination. First, the simplest option is to account for every possible cast in the sequence; we call this the *free* (F) cast reduction semantics. In particular this means that casts on constructors do not reduce and simply accumulate. Second, one can choose to accumulate intermediate casts using a special construct called a *meeting* (M) to store the accumulated meet (i.e., greatest lower bound with respect to precision) of the indices of the targets of the casts.[1] This technique has the advantage of ensuring cast space efficiency, as any cast sequence is represented in a single meeting, faithfully respecting every cast in the original sequence. A third option, which is also space efficient, is to drop intermediate casts and just keep the outermost one. This semantics is called *forgetful* (U) [Greenberg 2017] because it forgets (possibly invalid) intermediate casts. Here is an example of the expected cast reduction rules for the three semantics, where the empty vector `nil` of type $\mathbb{V}\,A\,0$ is first cast to $\mathbb{V}\,A\,1$ and then to its original type:

$$\begin{array}{rcll} & \not\rightsquigarrow & \text{(normal form)} & (F) \\ \langle \mathbb{V}\,A\,0 \Leftarrow \mathbb{V}\,A\,1 \Leftarrow \mathbb{V}\,A\,0 \rangle\, \texttt{nil} & \rightsquigarrow & \langle \mathbb{V}\,A\,0 \Leftarrow \mathbb{V}\,A\ \texttt{err}_{\mathbb{N}} \Leftarrow \mathbb{V}\,A\,0 \rangle\, \texttt{nil} & (M) \\ & \rightsquigarrow & \langle \mathbb{V}\,A\,0 \Leftarrow \mathbb{V}\,A\,0 \rangle\, \texttt{nil} & (U) \end{array}$$

We observe that these cast reduction semantics can be characterized in terms of *precision* ($\sqsubseteq$) through the interaction of the eliminator (called `catch`) and casts. This difference in behavior divides the space of possible semantics for indexed inductive types into three distinct classes depending on

[1]This construct was previously known in the literature as a *threesome* [Siek and Wadler 2010]. In cooperation with the original authors, we choose to rebrand to avoid the double entendre. The new name *meeting* was suggested by Phil Wadler.

which property they satisfy. Inductive types satisfy *equiprecise elimination* when they commute perfectly (i.e., moving casts in and out of `catch` does not affect precision), *overprecise elimination* when they gain precision when moving casts to the outside of the `catch`, and *underprecise elimination* when they lose precision by doing so. For instance, the forgetful semantics exhibits an overprecise eliminator: casting a constructor before eliminating it may fail less than casting after eliminating. For example, let us consider a function $f : \Pi(A : \square)(n : \mathbb{N}), \mathbb{V}\, A\, n \rightarrow \texttt{if}\ (\texttt{is\_err}\ n)\ \texttt{then}\ \texttt{err}_{\square}\ \texttt{else}\ \mathbb{N}$ on forgetful vectors that returns $\texttt{err}_{\texttt{err}_{\square}}$ when the index n is $\texttt{err}_{\mathbb{N}}$ and 0 otherwise. Because forgetful inductives drop intermediate casts, casting `nil` before applying it to f is less precise than casting after its application instead. In the following example, the cast through $\mathbb{V}\, A\ \texttt{err}_{\mathbb{N}}$ is ignored and the reduction proceeds successfully:

$$f\,(\langle \mathbb{V}\, A\, 0 \Leftarrow \mathbb{V}\, A\ \texttt{err}_{\mathbb{N}} \Leftarrow \mathbb{V}\, A\, 0\rangle\ \texttt{nil}) \rightsquigarrow f\,(\langle \mathbb{V}\, A\, 0 \Leftarrow \mathbb{V}\, A\, 0\rangle\ \texttt{nil}) \rightsquigarrow \langle \mathbb{N} \Leftarrow \mathbb{N}\rangle\,(f\ \texttt{nil})$$
$$\rightsquigarrow \langle \mathbb{N} \Leftarrow \mathbb{N}\rangle\, 0 \rightsquigarrow 0$$

However, when casting on the motive after applying f, the cast does go through $\texttt{err}_{\square}$ and fails:

$$\langle \mathbb{N} \Leftarrow \texttt{err}_{\square} \Leftarrow \mathbb{N}\rangle\,(f\ \texttt{nil}) \rightsquigarrow \langle \mathbb{N} \Leftarrow \texttt{err}_{\square} \Leftarrow \mathbb{N}\rangle\, 0 \rightsquigarrow \texttt{err}_{\mathbb{N}}$$

To explore these different semantics in a unified way we devise the notion of an *index accumulator*, a data structure used to accumulate the indices of the targets when casting terms of an indexed inductive types. The three examples above can be interpreted using different index accumulators: the free semantics can be interpreted using a list to accumulate every index, the meet semantics can be interpreted using a pair holding the accumulated precision meet [Siek and Wadler 2010] together with the outermost index, and the forgetful semantics can be interpreted using an option type to record the index of the outermost cast, if any. Rewriting the previous examples using index accumulators (in blue) we get the following:

$$\begin{array}{rcll} & \rightsquigarrow & \texttt{nil}\ [0, 1] & (F) \\ \langle \mathbb{V}\, A\, 0 \Leftarrow \mathbb{V}\, A\, 1 \Leftarrow \mathbb{V}\, A\, 0\rangle\ \texttt{nil} & \rightsquigarrow & \texttt{nil}\ (0, \texttt{err}_{\mathbb{N}}) & (M) \\ & \rightsquigarrow & \texttt{nil}\ (\text{some}\ 0) & (U) \end{array}$$

Apart from the way that indices of casts are accumulated, all semantics share most of their typing and reduction rules. We abstract indexed inductive definitions into a framework, called Punk, parametrized by an abstract index accumulator. We give a formal definition of index accumulators as a type inhabited by a distinguished term that represents the absence of casts and an action over indices used to accumulate them. For instance, the free index accumulator is a list of indices, where the distinguished term is the empty list and the action is appending an index to the head of the list. Precision on index accumulators must reflect the fact that they are used to accumulate casts and therefore should internalize properties of precision such as monotonicity and retraction [Maillard et al. 2022]. The index accumulator abstraction enables us to prove general properties, such as subject reduction and graduality, for any concrete semantics of gradual indexed inductive types, as long as its corresponding index accumulator satisfies certain properties.

As indexed inductive types are widely used in both theorem proving and dependently-typed programming, this work is an important step in the quest for making gradual theorem provers and gradual dependently-typed programming languages a reality.

Contributions. This work makes the following contributions:

- We explore the design space of indexed inductive types, identify various semantics and characterize them in three different classes, based on the interaction between elimination and casts, in terms of precision (§ 3).
- We propose the Punk framework, an extension of GRIP with gradual indexed inductive types, supporting multiple cast semantics (§ 5).

- We formally establish the metatheory of Punk and provide instances for the three classes of inductive families (§6).

2 Background on Gradual Dependent Type Theory

In this section we give an overview on the work that this articles builds upon, its base gradual type theory and some previous approaches to gradual indexed inductive types.

2.1 The Gradualized Calculus of Inductive Constructions (GCIC)

Lennon-Bertrand et al. [2022] introduced GCIC, an extension of the Calculus of Inductive Constructions (CIC), which incorporates gradual typing to the language. Like in a gradually typed lambda calculus, GCIC has an unknown type $?_\square$, optimistically representing any type, and an ascription operator $t :: B$ that allow to treat a term t as if it had type B regardless of the actual type of t. This permits the construction of terms like $u \triangleq (\lambda(x : ?_\square).\mathsf{not}\, x)\, 0$ that, although optimistically well-typed, informally correspond to a run-time failure that dynamically enforce typing invariants.

In GCIC, the semantics are defined by elaboration into a cast calculus named CastCIC. This calculus extends CIC, introducing two exceptional terms $\mathtt{err}_A$ and $?_A$ that represent runtime errors and unknown terms for any given type A, respectively. Additionally, CastCIC introduces a casting operator $\langle B \Leftarrow A \rangle\, t$ coercing a term t of type A to a term of type B. As an example, the previous GCIC expression u elaborates in CastCIC to $(\lambda(x : ?_\square).(\mathsf{not}\, \langle \mathbb{B} \Leftarrow ?_\square \rangle\, t))\, (\langle ?_\square \Leftarrow \mathbb{N} \rangle\, 0)$. Casts in CastCIC then reduce by comparing their source and target type, failing dynamically when a mismatch is found. In the running example, after β-reducing the top-most redex, we obtain $\mathsf{not}\, \langle \mathbb{B} \Leftarrow ?_\square \rangle\, (\langle ?_\square \Leftarrow \mathbb{N} \rangle\, 0)$ that reduces to $\mathsf{not}\, \mathtt{err}_{\mathbb{B}}$ because 0 is not a canonical form of type $\mathbb{B}$. A gradual dependent type system introduces a new challenge since computations involving exceptional terms may appear in types.

CastCIC features a single hierarchy of universes, denoted as Type or $\square$. The calculus supports simple inductive types like $\mathbb{N}$ or $\mathbb{B}$, as well as parameterized types like list. However, it does not offer a generalized support for indexed inductive types. In CastCIC, every type includes exceptional terms in the form of runtime errors (err) and unknown terms (?), with the following typing rules:

$$\frac{\Gamma \vdash A : \square_\ell}{\Gamma \vdash \mathtt{err}_A : A} \qquad\qquad \frac{\Gamma \vdash A : \square_\ell}{\Gamma \vdash ?_A : A}$$

These exceptional terms follow a *call-by-name* semantics [Pédrot and Tabareau 2018], implying that errors and unknown terms do not propagate unless they are forced in a computation. For example, $(\lambda(x : \mathbb{N}).0)\, \mathtt{err}_{\mathbb{N}}$ reduces to 0 rather than propagating $\mathtt{err}_{\mathbb{N}}$. Exceptional terms are considered canonical forms for positive types, while for negative types, they propagate exceptions when observed. For instance $\mathtt{err}_{\mathbb{N}\to\mathbb{N}}$ reduces to $(\lambda(x : \mathbb{N}).\, \mathtt{err}_{\mathbb{N}})$.

Furthermore, CastCIC includes the previously mentioned cast operator $\langle B \Leftarrow A \rangle\, t$, allowing terms of one type to be interpreted as another. The typing rule for casts its the expected one:

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

Throughout this paper we use the notation $\langle C \Leftarrow B \Leftarrow A \rangle\, t$ as a shorthand for two consecutive casts $\langle C \Leftarrow B \rangle\, \langle B \Leftarrow A \rangle\, t$.

The fire triangle of graduality. Lennon-Bertrand et al. [2022] study three properties, apart from subject reduction, that can be expected from a gradual type theory: normalization, conservativity with respect to a theory, and graduality. Normalization ensures the logical consistency of the system and is a desirable property for any type theory, especially when used as the underlining theory of a theorem prover. The other two properties are part of the *gradual guarantees* [Siek et al. 2015a], a group of properties expected from every gradually -typed language. Conservativity with respect to a static system states that the semantics of the gradual and static systems coincide on their common terms. Finally, graduality [New and Ahmed 2018] ensures that dynamic checks, such as casts, should only perform type checking: successful casts cannot affect the behavior of a program and invalid ones should fail more. However, the *Fire Triangle of Graduality* [Lennon-Bertrand et al. 2022] states that any gradual type theory cannot simultaneously satisfy the three properties. Consequently, GCIC and CastCIC come in three variants, each one sacrificing one property.

2.2 Internal Notion of Precision

Maillard et al. [2022] present GRIP, an extension of CastCIC$^{\mathcal{N}}$—the normalizing variant that sacrifices graduality as a global property—that introduces an internal representation of precision. This allows the user to reason about precision and graduality of terms within the type theory itself. To this end, they introduce an additional hierarchy of universes, denoted Prop, beside that of gradual types. Prop is a proof-irrelevant hierarchy of universes dedicated to *pure* propositions: propositions that can be *about* exceptional terms (`err` or ?) but cannot be *inhabited* by these exceptional terms. Prop hosts two key relations for internal reasoning about precision: *term precision* and *type precision*. Term precision, noted $a\ {}_{A}{\sqsubseteq}_{B}\ b$, states that the term a of type A is more precise than b of type B as terms, with the unknown term $?_B$ as a top element and $\mathtt{err}_A$ as a bottom element. Term precision is an heterogeneous relation, i.e., the types A and B are not required to be the same. Type precision, noted $A \sqsubseteq_\ell B$, states that A is more precise than B as types at universe level ℓ. This distinction circumvents the *Fire Triangle*: the unknown type is not a top element for type precision and term precision for types: $A\ {}_{\square_\ell}{\sqsubseteq}_{\square_\ell}\ B$ only holds when A is more precise than B as a term, and are bounded by the unknown type $?_{\square_\ell}$ (i.e., $A \sqsubseteq_\ell B \wedge B \sqsubseteq_\ell ?_{\square_\ell}$).

The precision relations live in the pure fragment of GRIP and therefore forbid *exceptional* proofs. Precision is a transitive relation, but unlike in more simple type systems, only the *gradual terms* (i.e., terms for which graduality holds) are related to themselves, or *self-precise*. In particular, functions are self-precise when they are *monotone* with respect to precision (i.e., they map related inputs to related outputs). We write $A^{\sqsubseteq}$ when a type A is self-precise at the level of types and $a^{\sqsubseteq_A}$ when a term a of type A is self-precise as a term. Precision cannot be reflexive in general because it would break conservativity of the system with respect to CIC, forbidding non-monotone functions, such as a dependently-typed `printf` function. Instead, precision is *quasi-reflexive*, meaning that if two terms are related by precision, then both are self-precise.

Since GRIP admits non-self-precise terms and can consistently reason about the precision of programs, the authors consider a general eliminator `catch` for inductive types that provide branches for exceptional inhabitants [Pédrot et al. 2019]. These exceptional branches allow the user to handle exceptions as they see fit. Therefore, `catch` makes it possible to define non-monotone functions: e.g., consider $f : \mathbb{B} \to \mathbb{B}$ that matches on a boolean, swaps `err` and ? (i.e., $f\ \mathtt{err}_{\mathbb{B}} \rightsquigarrow ?_{\mathbb{B}}$ and $f\, ?_{\mathbb{B}} \rightsquigarrow \mathtt{err}_{\mathbb{B}}$) and is the identity on pure booleans. However, `catch` is monotone whenever each constructor branch falls between the error branch and the unknown branch in terms of precision [Maillard et al. 2022]. Under mild assumptions on the precision of the motive and branches, a monotone eliminator `elim` not mentioning the exceptional cases can then be systematically derived from `catch` by forwarding the exceptions.

2.3 Basics of Indexed Inductive Types

Now we turn our focus to the main subject of this paper, namely indexed inductive types. In simple terms, indexed inductive types are just a *shape* plus an *index*. By shape we mean the basic

```
Fixpoint 𝕍fix (A : □) (n : ℕ) : □ := catch n with          Definition 𝕍eq (A : □) (n : ℕ) : □ :=
| 0    ⇒ ⊤              | errℕ ⇒ err□                        { l : list A | length l = n }
| S m ⇒ A * 𝕍fix A m    | unkℕ ⇒ unk□ end.
```

Fig. 1. Encodings of indexed inductive types in Coq

structure of inhabitants of the type, for example, the shape of `list` ℕ, lists of natural numbers, is a finite (thanks to the constructor `nil`) sequence of numbers (using the constructor cons). The constructors completely describe the shape of their inductive type, meaning that every closed term of the inductive must be convertible to one of its constructors. This enables the use of case analysis to eliminate the terms of an inductive type and more importantly it induces an induction principle to establish properties of these terms. Indices, on the other hand, constrain the terms inhabiting an indexed inductive type, encoding properties and invariants of the inductive data. For example, we can encode sized list of A as a type 𝕍 A n indexed by a natural number n where constructors are annotated with their size: the empty list `nil` has length 0 and cons'ing a new element on top of a list increases the length of the list by 1. Indices are specially useful to eliminate impossible cases when reasoning using induction. For instance, we can define the function hd in Coq that extracts the head of a vector with at least one element like this:

```
Definition hd (A : □) (n : ℕ) (v : 𝕍 A (S n)) : A := match v with | cons _ a _ ⇒ a end.
```

This function is defined by case analysis on the vector argument v; if the vector is a cons it returns its content, but if the vector is empty (i.e., it is `nil`) there is no content to extract! The latter case does not need to be considered since the type of the function makes it impossible to apply it to an empty vector.

Incorporating explicit support for indexed inductive types in a type theory that already accommodates standard inductive types is not strictly essential for their definition. This can be achieved by leveraging the existing features within the system (e.g., see $\mathbb{V}_{\text{eq}}$ in Figure 1). However, this approach can result in reduced logical strength, leading to diminished expressiveness and the introduction of potential inconsistencies. For instance, without the capability for large elimination—namely, the ability to construct a type depending on the structure of an inductive type—there are certain types and functions related to these types that cannot be adequately expressed (e.g., see $\mathbb{V}_{\texttt{fix}}$ in Figure 1). For instance, the absence of large elimination precludes the definition of a type family over booleans that is inhabited exclusively in the `true` fiber. Furthermore, this limitation extends to function definitions; specifically, the hd function cannot be defined, as its correct application necessitates a 𝕍 with a length strictly greater than zero to ensure subject reduction.

Previous approaches to indexed inductive types in CastCIC. Although Lennon-Bertrand et al. [2022] presents a way to define non indexed inductive types and indexed inductive types with forceable indices, they argue that extending CastCIC with a general notion of indexed inductive types is not trivial. Their approach is to extend indexed inductive type definitions with extra constructors that represent casts to the unknown index. For instance, the definition of the type 𝕍 for lists indexed by their length is extended with two new constructors nil? representing $\langle \mathbb{V}\, A\, ?_{\mathbb{N}} \Leftarrow \mathbb{V}\, A\, 0\rangle\, \texttt{nil}$ and cons? representing $\langle \mathbb{V}\, A\, ?_{\mathbb{N}} \Leftarrow \mathbb{V}\, A\, (\mathsf{S}\, n)\rangle\, \mathsf{cons}\, n\, x\, xs$. Cast on constructors to imprecise indices reduce to their imprecisely indexed counterparts and vice versa on cast from imprecise indices.

$$\langle \mathbb{V}\, A\, ?_{\mathbb{N}} \Leftarrow \mathbb{V}\, A\, 0\rangle\, \mathsf{nil} \rightsquigarrow \mathsf{nil?} \qquad\qquad \langle \mathbb{V}\, A\, 0 \Leftarrow \mathbb{V}\, A\, ?_{\mathbb{N}}\rangle\, \mathsf{nil?} \rightsquigarrow \mathsf{nil}$$

This approach does not generalize to arbitrary indices because the reduction rules discriminate the indices of the cast, here $?_{\mathbb{N}}$ and 0. Indices taken in a function type or an arbitrary type, like in the case of propositional equality, cannot be discriminated in this fashion.

Eremondi et al. [2022] solves this problem for propositional equality $x =_A y$, by extending refl with a new argument w, more precise than both indices $x, y : A$, that serves as a witness of the casts that have been performed on the equality proof. In this approach, casts on refl are performed without matching on the target's index and are instead accumulated in the witness using the meet of the indices of the casts. For this reason, they define an operator, called composition, that computes the meet with respect to precision of any two terms. This operation is defined for every type, even dependent products.

The elimination of propositional equality via the J rule performs the substitution as usual and then casts the result to and from the predicate applied to the witness. If M is a predicate over A, a and b two terms of type A, t an instance of $M\,a$, and refl_w a proof of $a =_A b$ with witness w, the reduction for J is defined as follows:

$$\mathsf{J}(A, x.M\,x, a, b, t, \mathsf{refl}_w) \rightsquigarrow \langle M\,b \Leftarrow M\,w\rangle\,\langle M\,w \Leftarrow M\,a\rangle\,t$$

Using propositional equality it is possible to define indexed inductive types out of parametrized inductive types using *fording*,[2] that is, using a proof of equality to enforce that the constructors have the correct index. For instance, in the definition for vectors the index $\mathbb{N}$ becomes a parameter $(n : \mathbb{N})$ and the type of each constructor is now guarded by an equality constraint on n:

```
Inductive 𝕍 (A : □) (n : ℕ) : □ :=
| nil : n = 0 → 𝕍 A n
| cons : ∀ (m : ℕ), n = S m → A → 𝕍 A m → 𝕍 A n.
```

Note that this approach requires the definition of inductive types with *non-uniform parameters* (i.e., parameters that can vary in recursive occurrences of the inductive type). Moreover, this approach reflects the cast reduction semantics of propositional equality. In this paper, we explore multiple cast reduction semantics and provide a direct approach instead of relying on this encoding.

3 Overview of Gradual Indexed Inductive Types and Punk

Since previous approaches to gradual indexed inductive types have shortcomings, we pursue here a general and proper treatment of these suitable for a gradual proof assistant. In this section we give a high level presentation of our solution for supporting indexed inductive types in a gradual type theory. In §3.1, we address the question of encoding gradual indexed inductive types using features already available in type theory, exhibiting their limitations, and devise well-behavedness criterion in §3.2. Then, we explore some possible semantics for gradual indexed inductive types (§3.3) and finish by giving an overview of our solution (§3.4).

3.1 Encodings Are Not Satisfactory

Fixpoint encoding. Inductive types indexed on inductive indices sometimes admit an encoding as a type level fixpoint in a non-gradual type theory [Brady et al. 2004]. In that situation, the type family is defined as a Type-valued recursive function by induction on indices. For example, the fixpoint encoding of vectors $\mathbb{V}_{\texttt{fix}}$ (Figure 1) is defined by mapping the index 0 to the unit type $\top$ with a single element for the argument-less constructor `nil`, and $\mathsf{S}\,n$ to the product $A \times \mathbb{V}_{\texttt{fix}}\,A\,n$ for `cons`, which takes an element of type A and a vector of length n to produce a vector of length $\mathsf{S}\,n$.

However, in a gradual setting this encoding admits more terms than one would expect on unknown indices. For instance, the type $\mathbb{V}_{\texttt{fix}}\,A\,?_{\mathbb{N}}$ is convertible to the unknown type $?_\square$: any term of any type can be cast to a valid vector of unknown index (e.g., $\langle ?_\square \Leftarrow \mathbb{N}\rangle\,0 : \mathbb{V}_{\texttt{fix}}\,\mathbb{B}\,?_{\mathbb{N}}$). The introduction of these extra inhabitants break the expected shape of the type, since it will be

[2]This technique was invented by Coquand for his work on the proof assistant ALF in the 1990s, as a way to reduce indexed inductive types to parametrized inductive types and an equality type. The name fording was first coined by McBride [1999].

inhabited by constructors other than `nil` and `cons`. On the other hand, `nil` and `cons` are still the only non exceptional canonical forms with indices 0 and $\mathsf{S}\,n$, respectively, thus respecting the constraints enforced by the indices.

Subset type encoding. Subset types $\{a : A \mid M\,a\}$ represent elements of a carrier type A satisfying some predicate $M : A \to \mathsf{Prop}$, that is the subset of the inhabitants a of A for which $M\,a$ holds. Terms of a subset type are tuples (a, p) consisting of a term $a : A$ and a proof $p : M\,a$.

In non-gradual type theories it is possible to encode an indexed inductive type as the subset of a non-indexed inductive type with the same shape whose elements satisfy the properties represented by their indices [Caglayan 2021]. For instance, vectors can be encoded as a list together with a proof that it has the correct size ($\mathbb{V}_{\mathsf{eq}}$ in Figure 1).

In contrast, subset types do not behave well in a gradual type theory. The core issue lies in the incompatibility of propositions with gradual operations. Essentially, any gradual proposition is inhabited by exceptional proofs making the system inconsistent and logically unsound: $\bot$ is trivially provable by $?_\bot$. Moreover, casts between gradual propositions would need to synthesize proofs on the fly. Since arbitrary proofs cannot be synthesized in a computable and total fashion, Maillard et al. [2022] define cast between gradual propositions to always reduce to an exception.

Therefore, gradual subset types cannot enforce properties over terms, and the corresponding encoding of indexed inductive types does not maintain the invariants associated to the indices across casts. For instance, it would be easy to construct an empty vector with index 1 as $(\mathtt{nil}, \mathtt{err}_{\mathtt{length\,nil=1}})$. Since, using this encoding, the index of an inductive type says nothing about its inhabitants, indexed inductive types degenerate into regular non-indexed inductive types (e.g., vectors become equivalent to lists).

3.2 Requirements for Gradual Indexed Inductive Types

Informally, the encodings presented in the previous section do not provide the invariants expected from indexed inductive types. We identify two criteria to guarantee that an implementation of gradual indexed inductive types do not exhibit these issues. These criteria, *index relevance* and *shape relevance*, correspond to the two key characteristics of indexed inductive types, their shape and index.

Definition 3.1 (Index relevance). A self-precise type family $X : A \to \square_{\ell'}$ over $A : \square_\ell$ is index relevant when precision-related instances of X imply relatedness of the indices:

$$\forall (a, a' : A).\, X\,a \sqsubseteq_{\ell'} X\,a' \to a \;{}_A{\sqsubseteq}_A\; a'$$

In categorical language, X is full and faithful, when viewed as a functor from the precision preorder on the domain A to the type precision preorder.

For instance, the inductive family $\mathbb{V}$ of vectors is index relevant if $\mathbb{V}\,A\,m \sqsubseteq_\ell \mathbb{V}\,A\,n$ implies $m \;{}_{\mathbb{N}}{\sqsubseteq}_{\mathbb{N}}\; n$. As we saw in §3.1, encoding indexed inductive types as subset types cannot be index relevant.

Definition 3.2 (Shape relevance). A self-precise type family $X : A \to \square_{\ell'}$ over $A : \square_\ell$ is shape relevant if its image $\{X\,a\}_{a:A} : \square_{\ell'}$ in the universe is both upward and downward closed with respect to precision:

$$\forall (a : A).\, a^{\sqsubseteq_A} \to T \sqsubseteq_\ell X\,a \to (\exists (a' : A), T = X\,a') \lor (T = \mathtt{err}_{\square_{\ell'}})$$
$$\forall (a : A).\, a^{\sqsubseteq_A} \to X\,a \sqsubseteq_\ell T \to (\exists (a' : A), T = X\,a') \lor (T = ?_{\square_{\ell'}})$$

Put simply, an inductive is shape relevant if it is only related by precision with the error type $\mathtt{err}_\square$, the unknown type $?_\square$ and itself, possibly applied to different parameters and indices. This criterion excludes degenerated cases such as $\mathbb{N} \sqsubseteq_\ell \mathbb{V}\,A\,?_{\mathbb{N}}$ and ensures that inductive types are

```
Inductive vec (A : □) : ℕ → □ :=
| nil : vec A 0
| cons : ∀ (n : ℕ), A
      → vec A n → vec A (S n).
```

```
Fixpoint len {A n} (v : vec A (S n)) : ℕ :=
catch v with
| nil          ⇒ 0           | err ⇒ errℕ
| cons _ _ v   ⇒ S (len v)   | unk ⇒ errℕ end.
```

```
Definition f_ty (n : ℕ) : □ :=
catch n with
| 0    ⇒ list ℕ   | errℕ ⇒ err□
| S _  ⇒ list 𝔹   | unkℕ ⇒ err□ end.
```

```
Definition f {A n} (v : 𝕍 A n) : f_ty n :=
catch v with
| nil           ⇒ [ 0 ]      | err ⇒ err
| cons _ _ _    ⇒ [ true ]   | unk ⇒ unk end.
```

Fig. 2. Definitions illustrating the behaviors of cast on GIIT in Gallina (Coq).

not inhabited by canonical forms of other types, a condition violated by fixpoint encodings (§3.1). Conversely, the approaches to gradual indexed inductive types discussed in §2.3 satisfy both criteria. We consider essential to ensure that the definition of well-behaved inductives encompasses the prior research on gradual indexed inductive types.

Apart from these two criteria, we also need to ensure that the gradual indexed inductive types satisfy the relevant gradual properties, in particular inductive type constructors must be self-precise. Following Maillard et al. [2022], self-precision for a type family $X : A \to \square$ mandates that any two elements $a, a' : A$ related by precision $a \;{}_A{\sqsubseteq}_A\; a'$ yield precision-related types $X\,a \sqsubseteq_\ell X\,a'$, that in turn need to preserve *graduality* [New and Ahmed 2018].

Definition 3.3 (Graduality). Cast between types related by precision $X \sqsubseteq_\ell Y$ induce an *embedding-projection pair*, meaning a Galois connection $\langle Y \Leftarrow X\rangle \dashv \langle X \Leftarrow Y\rangle$ with respect to precision such that $\langle Y \Leftarrow X\rangle \circ \langle X \Leftarrow Y\rangle \mathrel{\sqsupseteq\!\sqsubseteq} \mathrm{id}_X$.

3.3 Possible Semantics for Gradual Indexed Inductive Types

As we saw in § 1, there are many valid semantics for gradual indexed inductive types, and it is not clear if one is better than the other. These different inductive definitions can be classified according to their eliminator's interaction with casts, up to precision. Since precision is asymmetric, we identify three primary behaviors: preservation of precision (equiprecise elimination), gain in precision (underprecise elimination), and loss of precision (overprecise elimination), explicitly excluding the case when there is no relation. In the following, we explore how inductive types from each of these classes behave, illustrated with examples. In each example we observe the behavior of the three semantics for vectors illustrated in the introduction.

3.3.1 Eliminating with a Constant Motive. We start with a simple example: computing the length of the vector $\langle \mathbb{V}\,\mathbb{B}\,0 \Leftarrow \mathbb{V}\,\mathbb{B}\,1 \Leftarrow \mathbb{V}\,\mathbb{B}\,0\rangle\,\mathtt{nil}$ using the function `len` provided in Figure 2.

Free vectors. Casts on free vectors do not reduce, delegating the task of evaluating casts to the result of an elimination:

$$\mathtt{len}\,(\langle \mathbb{V}\,\mathbb{B}\,0 \Leftarrow \mathbb{V}\,\mathbb{B}\,1 \Leftarrow \mathbb{V}\,\mathbb{B}\,0\rangle\,\mathtt{nil}) \rightsquigarrow \langle \mathbb{N} \Leftarrow \mathbb{N} \Leftarrow \mathbb{N}\rangle\,(\mathtt{len}\,\mathtt{nil}) \rightsquigarrow \langle \mathbb{N} \Leftarrow \mathbb{N} \Leftarrow \mathbb{N}\rangle\,0 \rightsquigarrow 0$$

Meet vectors. Casts on meet vectors reduce to a meeting and then propagates the casts:

$$\begin{aligned}\mathtt{len}\,(\langle \mathbb{V}\,\mathbb{B}\,0 \Leftarrow \mathbb{V}\,\mathbb{B}\,1 \Leftarrow \mathbb{V}\,\mathbb{B}\,0\rangle\,\mathtt{nil}) &\rightsquigarrow \mathtt{len}\,(\langle \mathbb{V}\,\mathbb{B}\,0 \Leftarrow \mathbb{V}\,\mathbb{B}\,\mathtt{err}_{\mathbb{N}} \Leftarrow \mathbb{V}\,\mathbb{B}\,0\rangle\,\mathtt{nil})\\ &\rightsquigarrow \langle \mathbb{N} \Leftarrow \mathbb{N} \Leftarrow \mathbb{N}\rangle\,(\mathtt{len}\,\mathtt{nil}) \rightsquigarrow \langle \mathbb{N} \Leftarrow \mathbb{N} \Leftarrow \mathbb{N}\rangle\,0 \rightsquigarrow 0\end{aligned}$$

Forgetful vectors. Forgetful vectors ignore intermediate casts and so maintain at most one cast:

$$\begin{aligned}\mathtt{len}\,(\langle \mathbb{V}\,\mathbb{B}\,0 \Leftarrow \mathbb{V}\,\mathbb{B}\,1 \Leftarrow \mathbb{V}\,\mathbb{B}\,0\rangle\,\mathtt{nil}) &\rightsquigarrow \mathtt{len}\,(\langle \mathbb{V}\,\mathbb{B}\,0 \Leftarrow \mathbb{V}\,\mathbb{B}\,0\rangle\,\mathtt{nil})\\ &\rightsquigarrow \langle \mathbb{N} \Leftarrow \mathbb{N}\rangle\,(\mathtt{len}\,\mathtt{nil}) \rightsquigarrow \langle \mathbb{N} \Leftarrow \mathbb{N}\rangle\,0 \rightsquigarrow 0\end{aligned}$$

It is interesting to note that because the motive of the elimination ($\lambda\_\,\_.\mathbb{N}$) does not depend on the index, the three semantics are equivalent.

3.3.2 Eliminating with a Dependent Motive. Let us examine a more interesting example. The function f (Figure 2) operates on vectors with a return type f_ty that actually depends on the vector index. This function maps 0 to $\mathtt{list}\,\mathbb{N}$, $\mathsf{S}\,n$ to $\mathtt{list}\,\mathbb{B}$, and exceptional naturals $\mathtt{err}_{\mathbb{N}}$ and $?_{\mathbb{N}}$ to their corresponding exceptional types $\mathtt{err}_{\square}$ and $?_{\square}$. The function f transforms the empty vector nil to a singleton list [0], cons vectors to [true], and propagates exceptional terms. Recall that in exceptional type theory, exceptions are by-name [Pédrot et al. 2019], meaning that terms may have exceptions as subterms. In particular, this means that $[\mathtt{err}_{\mathbb{N}}]$ is a normal form and does not reduce to $\mathtt{err}_{\mathtt{list}\,\mathbb{N}}$. Given that f's return type is dependent, applying our three vector semantics to f v reveals varied outcomes.

Free vectors. In the free semantics every cast is taken into account and, after propagating the casts outside of the eliminator, an intermediate cast through $\mathtt{list}\,\mathbb{B}$ remains. Since we only perform casts between lists, the shape of the list is preserved (i.e., it stays a singleton), but the inner cast fails because the parameters are incompatible, resulting in a singleton list with an error inside.

$$\begin{aligned}&\mathtt{f}\,\langle \mathbb{V}\,\mathbb{B}\,0 \Leftarrow \mathbb{V}\,\mathbb{B}\,1 \Leftarrow \mathbb{V}\,\mathbb{B}\,0\rangle\,\mathtt{nil}\\ &\quad\rightsquigarrow \langle \mathtt{list}\,\mathbb{N} \Leftarrow \mathtt{list}\,\mathbb{B} \Leftarrow \mathtt{list}\,\mathbb{N}\rangle\,(\mathtt{f}\,\mathtt{nil}) \rightsquigarrow \langle \mathtt{list}\,\mathbb{N} \Leftarrow \mathtt{list}\,\mathbb{B} \Leftarrow \mathtt{list}\,\mathbb{N}\rangle\,[0]\\ &\quad\rightsquigarrow \langle \mathtt{list}\,\mathbb{N} \Leftarrow \mathtt{list}\,\mathbb{B}\rangle\,[\langle \mathbb{B} \Leftarrow \mathbb{N}\rangle\,0] \rightsquigarrow \langle \mathtt{list}\,\mathbb{N} \Leftarrow \mathtt{list}\,\mathbb{B}\rangle\,[\mathtt{err}_{\mathbb{B}}] \rightsquigarrow [\mathtt{err}_{\mathbb{N}}]\end{aligned}$$

Meet vectors. Meet semantics consolidate intermediate casts into one, through the accumulated meet of their indices, here represented by $0 \sqcap 1 \rightsquigarrow \mathtt{err}_{\mathbb{N}}$. After propagating the casts out of the eliminator, the intermediate cast goes through $\mathtt{err}_{\square}$ because f_ty propagates the exception. In contrast to the free semantics, the structure of the list is not preserved.

$$\begin{aligned}&\mathtt{f}\,\langle \mathbb{V}\,\mathbb{B}\,0 \Leftarrow \mathbb{V}\,\mathbb{B}\,1 \Leftarrow \mathbb{V}\,\mathbb{B}\,0\rangle\,\mathtt{nil}\\ &\quad\rightsquigarrow \mathtt{f}\,\langle \mathbb{V}\,\mathbb{B}\,0 \Leftarrow \mathbb{V}\,\mathbb{B}\,\mathtt{err}_{\mathbb{N}} \Leftarrow \mathbb{V}\,\mathbb{B}\,0\rangle\,\mathtt{nil} \rightsquigarrow \langle \mathtt{list}\,\mathbb{N} \Leftarrow \mathtt{err}_{\square} \Leftarrow \mathtt{list}\,\mathbb{N}\rangle\,(\mathtt{f}\,\mathtt{nil})\\ &\quad\rightsquigarrow \langle \mathtt{list}\,\mathbb{N} \Leftarrow \mathtt{err}_{\square} \Leftarrow \mathtt{list}\,\mathbb{N}\rangle\,[0] \rightsquigarrow \langle \mathtt{list}\,\mathbb{N} \Leftarrow \mathtt{err}_{\square}\rangle\,\mathtt{err}_{\mathtt{err}_{\square}} \rightsquigarrow \mathtt{err}_{\mathtt{list}\,\mathbb{N}}\end{aligned}$$

Forgetful vectors. In the forgetful semantics the intermediate cast through $\mathbb{V}\,\mathbb{B}\,1$ is dropped allowing the computation to recover from what could have been an invalid state.

$$\begin{aligned}\mathtt{f}\,\langle \mathbb{V}\,\mathbb{B}\,0 \Leftarrow \mathbb{V}\,\mathbb{B}\,1 \Leftarrow \mathbb{V}\,\mathbb{B}\,0\rangle\,\mathtt{nil} &\rightsquigarrow \mathtt{f}\,\langle \mathbb{V}\,\mathbb{B}\,0 \Leftarrow \mathbb{V}\,\mathbb{B}\,0\rangle\,\mathtt{nil}\\ &\rightsquigarrow \langle \mathtt{list}\,\mathbb{N} \Leftarrow \mathtt{list}\,\mathbb{N}\rangle\,(\mathtt{f}\,\mathtt{nil}) \rightsquigarrow \langle \mathtt{list}\,\mathbb{N} \Leftarrow \mathtt{list}\,\mathbb{N}\rangle\,[0] \rightsquigarrow [0]\end{aligned}$$

Interestingly, the outcomes of these semantics are distinguished by their precision levels. Specifically, meet vectors yield more precise results than free vectors, which in turn are more precise than those from forgetful semantics: $\mathtt{err}_{\mathtt{list},\mathbb{N}} \sqsubseteq [\mathtt{err}_{\mathbb{N}}] \sqsubseteq [0]$.

This illustrates the fact that each of these semantics belongs to a unique class of inductive types—free vectors to equiprecise elimination, meet vectors to underprecise elimination, and forgetful vectors to overprecise elimination. Importantly, this classification extends beyond vectors, applying to all indexed inductive types and highlighting the nuanced behavior of these three semantic approaches.

3.4 Overview of Punk

As we have discussed in the previous sections, introducing indexed inductive types to a gradual type theory requires us to think very carefully about how casts interact with constructors and the eliminator. Moreover, there is not a unique, nor best, behavior for cast reduction. To capture this fact, we present a framework for defining indexed inductive types that is parameterized by the way cast reduction works.

We describe different cast reduction semantics by defining a notion of *index accumulator*, a type used to accumulate the indices of the casts that have been applied to a constructor. Essentially, elements of an index accumulator correspond to a sequence of indices. An indexed accumulator is equipped with a term representing the empty accumulator $\mathbb{1}$ and an operation $i \otimes acc$ to extend an accumulator acc with a new index i. We extend the syntax of constructors with a new argument, the index accumulator. A constructor that has not been casted yet carries an empty accumulator and casts on a constructor extend the accumulator with the target index. In contrast to indices, casting the parameters cannot fail since they are shared by all constructors and require no special treatment: changes to parameters are propagated inside the constructor and applied to its arguments. For instance, casting the vector $w \triangleq \texttt{cons}\ \mathbb{1}\ \texttt{true}\ (\texttt{nil}\ \mathbb{1})$, reduces as follows:

$$\begin{aligned}\langle \mathbb{V}\,\mathbb{N}\,2 \Leftarrow \mathbb{V}\,\mathbb{B}\,1\rangle\, \texttt{cons}\ \mathbb{1}\ \texttt{true}\,(\texttt{nil}\ \mathbb{1}) &\rightsquigarrow \texttt{cons}\ (2\otimes\mathbb{1})\,(\langle\mathbb{N}\Leftarrow\mathbb{B}\rangle\texttt{true})\,(\langle\mathbb{V}\,\mathbb{N}\,0\Leftarrow\mathbb{V}\,\mathbb{B}\,0\rangle(\texttt{nil}\ \mathbb{1}))\\ &\rightsquigarrow \texttt{cons}\ (2\otimes\mathbb{1})\ \texttt{err}_{\mathbb{N}}\,(\texttt{nil}\ (0\otimes\mathbb{1}))\end{aligned}$$

A gradual indexed inductive type $\mathbb{I}\ p\ i$, as every other gradual type, has two extra inhabitants—the exceptional terms $\texttt{err}_{\mathbb{I}\,p\,i}$ and $?_{\mathbb{I}\,p\,i}$. To handle the elimination of these extra inhabitants, we borrow the `catch` eliminator from exceptional type theory [Pédrot et al. 2019]. This eliminator extends the eliminator from CIC with an extra branch for each exceptional term. In the case of Punk, `catch` has two new branches: $h_{\texttt{err}}$ and $h_{\texttt{unk}}$ for `err` and ?, respectively.

To handle the elimination of constructors using `catch`, index accumulators provide a representation of their elements as list of indices. When eliminating a constructor, the accumulated indices are propagated outside of the eliminator as casts and the elimination proceeds as if the accumulator was empty. For example, if the accumulator $2\otimes\mathbb{1}^p$ yield the list [2], applying f (Figure 2) on the example vector w reduces as follows:

$$\begin{aligned}\texttt{f}\,(\texttt{cons}\ (2\otimes\mathbb{1})\ \texttt{err}_{\mathbb{N}}\,(\texttt{nil}\ (0\otimes\mathbb{1}))) &\rightsquigarrow \langle\texttt{f\_ty}\,2\Leftarrow\texttt{f\_ty}\,1\rangle\,\texttt{f}\,(\texttt{cons}\ \mathbb{1}\ \texttt{err}_{\mathbb{N}}\,(\texttt{nil}\ (0\otimes\mathbb{1})))\\ &\rightsquigarrow \langle\texttt{list}\,\mathbb{B}\Leftarrow\texttt{list}\,\mathbb{B}\rangle\,[\texttt{true}] \rightsquigarrow [\texttt{true}]\end{aligned}$$

3.4.1 Different Semantics in Punk. Using different accumulators we can construct indexed inductive types with different cast reduction semantics. We focus on three different semantics: free, meet, and forgetful inductives.

Free inductives. Free indexed inductives accumulate the indices of every cast using a list. Casting a constructor extends the accumulator with the target index as its head, keeping all casts into account.

$$\langle\mathbb{V}\,\mathbb{B}\,0\Leftarrow\mathbb{V}\,\mathbb{B}\,1\Leftarrow\mathbb{V}\,\mathbb{B}\,0\rangle\,(\texttt{nil}\ [\,])\rightsquigarrow\langle\mathbb{V}\,\mathbb{B}\,0\Leftarrow\mathbb{V}\,\mathbb{B}\,1\rangle\,(\texttt{nil}\ [1])\rightsquigarrow\texttt{nil}\ [0,1]$$

Meet inductives. The meet semantics accumulates casts by storing the accumulated meet of the target indices of the casts applied to a constructor, together with the target index. Meet inductives accumulate indices using a type inhabited by the empty accumulator zero that represents the absence of casts and meet two i_1 i_2 with target index i_1 and accumulated meet i_2. On elimination, these constructors are interpreted as a list with 0 or 2 indices, respectively.

$$\begin{aligned}\langle\mathbb{V}\,\mathbb{B}\,0\Leftarrow\mathbb{V}\,\mathbb{B}\,1\Leftarrow\mathbb{V}\,\mathbb{B}\,0\rangle\,(\texttt{nil}\ \texttt{zero})&\rightsquigarrow\langle\mathbb{V}\,\mathbb{B}\,0\Leftarrow\mathbb{V}\,\mathbb{B}\,1\rangle\,(\texttt{nil}\ (\texttt{two}\ 1\ (0\sqcap 1)))\\ &\rightsquigarrow\texttt{nil}\ (\texttt{two}\ 0\ (0\sqcap 1\sqcap 0))\end{aligned}$$

$\boxed{\Gamma \vdash \Delta\ \textit{tele}}$ Telescope Δ in context Γ $\boxed{\Gamma \vdash \sigma : \Delta}$ Substitution σ of the telescope Δ over context Γ

Name	Description	Well-formedness
$\mathbf{Param}$	Parameters of the inductive $\mathbb{I}$	$\vdash \mathbf{Param}\ \textit{tele}$
$\mathbf{Idx}^p$	Indices of the inductive $\mathbb{I}$	$\mathbf{Param} \vdash \mathbf{Idx}^p\ \textit{tele}$
$\mathbf{\#ctors}$	Number of constructors of the inductive $\mathbb{I}$	$\mathbf{\#ctors} \in \mathbb{N}$
$\mathbf{Arg}^p_{\mathbb{c}}$	Arguments of the constructor $\mathbb{c}$	$\mathbf{Param} \vdash \mathbf{Arg}^p_{\mathbb{c}}\ \textit{tele}$
$\mathbf{\#indargs}_{\mathbb{c}}$	Number of inductive arguments of $\mathbb{c}$	$\mathbf{\#indargs}_{\mathbb{c}} \in \mathbb{N}$
$\mathbf{IndArg}^p_{\mathbb{c},k}$	Arguments of the k-th inductive argument of $\mathbb{c}$	$\mathbf{Param}, \mathbf{Arg}^p_{\mathbb{c}} \vdash \mathbf{IndArg}^p_{\mathbb{c},k}\ \textit{tele}$
$\mathbf{arg\_idx}^p_{\mathbb{c},k}$	Indices of the k-th inductive argument of $\mathbb{c}$	$\mathbf{Param}, \mathbf{Arg}^p_{\mathbb{c}}, \mathbf{IndArg}^p_{\mathbb{c},k} \vdash \mathbf{arg\_idx}^p_{\mathbb{c},k} : \mathbf{Idx}^p$
$\mathbf{ctr\_idx}^p_{\mathbb{c}}$	Indices of the constructor $\mathbb{c}$	$\mathbf{Param}, \mathbf{Arg}^p_{\mathbb{c}} \vdash \mathbf{ctr\_idx}^p_{\mathbb{c}} : \mathbf{Idx}^p$

Fig. 3. Defining components for an indexed inductive type

$$\mathbf{Param}_{\mathbb{V}} \triangleq (A : \square_\ell) \qquad \mathbf{Idx}^A_{\mathbb{V}} \triangleq (n : \mathbb{N}) \qquad \mathbf{\#ctors}_{\mathbb{V}} \triangleq 2 \qquad \mathbf{Arg}^A_{\mathbb{V},\mathtt{nil}} \triangleq ()$$

$$\mathbf{Arg}^A_{\mathbb{V},\mathsf{cons}} \triangleq (n : \mathbb{N})(a : A) \qquad \mathbf{\#indargs}_{\mathbb{V},\mathtt{nil}} \triangleq 0 \qquad \mathbf{\#indargs}_{\mathbb{V},\mathsf{cons}} \triangleq 1 \qquad \mathbf{IndArg}^A_{\mathbb{V},\mathsf{cons},1}\, n\, a \triangleq ()$$

$$\mathbf{arg\_idx}^A_{\mathbb{V},\mathsf{cons},1}\, n\, a \triangleq n \qquad \mathbf{ctr\_idx}^A_{\mathbb{V},\mathtt{nil}} \triangleq 0 \qquad \mathbf{ctr\_idx}^A_{\mathbb{V},\mathsf{cons}}\, n\, a \triangleq \mathsf{S}\, n$$

Fig. 4. Defining components for the type family of vectors.

Forgetful inductives. Forgetful inductives drop any intermediate cast and their accumulator holds only the target of the outermost cast if any. To this end, they accumulate indices using an option type where none stands for the absence of casts and some holds the index of its only cast when needed. Therefore the expected cast reduction rule for forgetful vectors is encoded as follows.

$$\langle \mathbb{V}\,\mathbb{B}\,0 \Leftarrow \mathbb{V}\,\mathbb{B}\,1 \Leftarrow \mathbb{V}\,\mathbb{B}\,0\rangle\,(\mathtt{nil}\ \mathsf{none}) \rightsquigarrow \langle \mathbb{V}\,\mathbb{B}\,0 \Leftarrow \mathbb{V}\,\mathbb{B}\,1\rangle\,(\mathtt{nil}\ (\mathsf{some}\ 1)) \rightsquigarrow \mathtt{nil}\ (\mathsf{some}\ 0)$$

The semantics that we explore in this paper use non-indexed inductive types as their index accumulators, however our framework is more general and it is not restricted to such types.

Summary. We saw that there is not a unique way to give semantics to gradual indexed inductive types. There is a degree of freedom regarding how casts commute with the eliminator up to precision and we have identified three meaningful alternatives. Each of these options gives rise to a class of indexed inductive types. Punk supports each of these classes with a specific instance: free, forgetful, and meet inductives. The semantic differences are captured by different choices of an index accumulator.

A note on language design. A specific gradual proof assistant can decide to either embrace a given accumulator semantics once and for all, or opt to allow programmers to configure this choice, either globally or on a case-by-case basis. In the latter option, the choice of the accumulator semantics can be made independently for each gradual indexed inductive type. In theory, one can even support two implementations of the same inductive family with different accumulator semantics; however, in that case, the two families are unrelated by precision, so any cast between them fails. As a formal framework designed to study these semantics, Punk is agnostic with respect to this issue.

$\boxed{\Gamma \vdash t : T}$

$$\textsc{$\mathbb{I}$-Ty}\quad \frac{\Gamma \vdash p : \mathbf{Param} \qquad \Gamma \vdash i : \mathbf{Idx}^{p}}{\Gamma \vdash \mathbb{I}\, p\, i : \square_{\ell}}$$

$$\textsc{$\mathbb{I}$-Ctor}\quad \frac{\begin{array}{c}\Gamma \vdash p : \mathbf{Param} \qquad \Gamma \vdash a : \mathbf{Arg}^{p}_{\mathbb{c}} \\ \forall\, j < \mathbf{\#indargs}_{\mathbb{c}},\ \Gamma \vdash r_j : \Pi(x_j : \mathbf{IndArg}^{p}_{\mathbb{c},j}\, a), \mathbb{I}\, p\, (\mathbf{arg\_idx}^{p}_{\mathbb{c},j}\, a\, x_j)\end{array}}{\Gamma \vdash \mathbb{c}^{p}\, a\, \overrightarrow{r_j} : \mathbb{I}\, p\, (\mathbf{ctr\_idx}^{p}_{\mathbb{c}}\, a)}$$

$$\textsc{$\mathbb{I}$-Elim}\quad \frac{\begin{array}{c}\Gamma \vdash p : \mathbf{Param} \qquad \Gamma \vdash M : \Pi(i : \mathbf{Idx}^{p}), \mathbb{I}\, p\, i \to \square_{\ell'} \\ \forall\, k < \mathbf{\#ctors},\ \Gamma \vdash h_{\mathbb{c}_k} : \mathsf{Branch}^{p}_{\mathbb{c}_k}\, M \qquad \Gamma \vdash i : \mathbf{Idx}^{p} \qquad \Gamma \vdash t : \mathbb{I}\, p\, i\end{array}}{\Gamma \vdash \mathtt{elim}^{p}\, M\, \overrightarrow{h_{\mathbb{c}_k}}\, i\, t : M\, i\, t}$$

$\boxed{t \rightsquigarrow t}$ $\boxed{\mathtt{canonical}\ t}$

$\textsc{$\mathbb{I}$-Elim-Ctor}$: $\mathtt{elim}^{p}\, M\, \overrightarrow{h_{\mathbb{c}_k}}\, i\, (\mathbb{c}^{p}\, a\, \overrightarrow{r_j}) \rightsquigarrow h_{\mathbb{c}}\, a\, \overrightarrow{r_j}\, \overrightarrow{t_j}$ $\qquad t_j \triangleq \lambda(x_j : \mathbf{IndArg}^{p}_{\mathbb{c},j}\, a), \mathtt{elim}^{p}\, M\, \overrightarrow{h_{\mathbb{c}_k}}\, (\mathbf{arg\_idx}^{p}_{\mathbb{c},j}\, a\, x_j)(r_j\, x_j)$

$$\textsc{Evaluation Contexts}\quad C ::= \ldots \mid \mathtt{elim}^{p}\, M\, \overrightarrow{h_{\mathbb{c}_k}}\, i\, C \qquad \textsc{Red-Cong}\quad \frac{t \rightsquigarrow t'}{C[t] \rightsquigarrow C[t']} \qquad \frac{}{\mathtt{canonical}\ \mathbb{c}^{p}\, a\, \overrightarrow{r_j}}$$

Fig. 5. Typing and reduction rules for static inductive types.

4 Syntax and Semantics of Indexed Inductive Families

General formal accounts of indexed inductive types are notationally demanding. In this section, we fix notations for indexed inductive types in CIC, and give their static and dynamic semantics. These notations are employed to present gradual indexed inductive types in subsequent sections.

An inductive definition specify a type constructor $\mathbb{I}$, term constructors $\mathbb{c}$ that serve as the canonical inhabitants of the inductive type, and an eliminator `elim` that consumes inductive terms of type $\mathbb{I}$ via case analysis with one branch for each of the inductive's term constructors.

The typing rules for static indexed inductive types (Figure 5) use a collection of metafunctions associated to the inductive definition (Figure 3).[3] Most metafunctions are parameterized by an instance p of the telescope of parameters **Param**. As an illustration, the components for the type family of vectors is shown in Figure 4.

Introduction. An inductive type family $\mathbb{I}$ is introduced as a type $\mathbb{I}\, p\, i$ using Rule $\mathbb{I}$-Ty with substitutions p and i for the telescopes of parameters and indices. A term of that type is formed through a constructor $\mathbb{c}^{p}\, p\, a\, \overrightarrow{r_j}$ (Rule $\mathbb{I}$-Ctor) where p is an instantiation of the parameters, a an instantiation of the non-inductive arguments and each r_j for $j \in \{0, \ldots, \mathbf{\#indargs}_{\mathbb{c}}\}$ is an inductive argument. The indices of the constructor are computed out of a and each inductive argument use the same instantiation of parameters but possibly different indices. Note that we force the inductive arguments to be strictly positive.

Elimination. The eliminator `elim` performs case analysis on a term of an indexed inductive type. It consist of a motive $M : \Pi(i : \mathbf{Idx}^{p}), \mathbb{I}\, p\, i \to \square$, the return type of the elimination; a method $h_{\mathbb{c}_k}$ for each constructor $\mathbb{c}_k$, that prescribes the behavior of `elim` on that constructor; and a term to be eliminated together with its index. Each branch take both the non-inductive and inductive arguments of the constructor, and then the induction hypotheses induced by these inductive arguments.

[3]Inductive definitions are typically carried in a typing signature Σ along the typing judgement and queried through the metafunctions. To simplify the notations we omit these signatures, and often let inductive annotation $\mathbb{I}$ implicit.

$$\begin{aligned}
\mathtt{Branch}^p_{\mathbb{c}}\, M \triangleq\; & \Pi(a : \mathbf{Arg}^p_{\mathbb{c}}) && \text{non-inductive arguments}\\
& (\forall\, j,\ (r_j : \Pi(x_j : \mathbf{IndArg}^p_{\mathbb{c},j}\, a), \mathbb{I}\, p\, (\mathbf{arg\_idx}^p_{\mathbb{c},j}\, a\, x_j))) && \text{inductive arguments}\\
& (\forall\, j,\ (\Pi(x_j : \mathbf{IndArg}^p_{\mathbb{c},j}\, a), M\, (\mathbf{arg\_idx}^p_{\mathbb{c},j}\, a\, x_j)\, (r_j\, x_j))), && \text{induction hypotheses}\\
& M\, (\mathbf{ctr\_idx}^p_{\mathbb{c}}\, a)\, (\mathbb{c}^p\, a\, \overrightarrow{r_j})
\end{aligned}$$

The constructors are the only canonical forms (Figure 5) and there is just one reduction rule: the elimination of constructors ($\mathbb{I}$-Elim-Ctor) that applies the correct branch and uses recursive calls to the eliminator as the induction hypotheses.

5 Punk: A Framework for Gradual Inductive Family Definitions

We describe a framework called Punk providing a schema for gradual indexed inductive types.[4] The reduction semantics of casts for each gradual indexed inductive type is derived uniformly from an index accumulator (§5.1). Section §5.2 provides instances of index accumulators to recover the free, forgetful and meet reduction semantics of casts illustrated in the introduction. The static and dynamic semantics are then derived in §5.3 for any choice of an index accumulator, as well as the extension of internal precision for gradual indexed inductive types (§5.4).

5.1 Index Accumulators

Definition 5.1 (Index Accumulator). An index accumulator $(\mathbf{Acc}^p, \otimes^p, \mathbb{1}^p)$ over a telescope of parameters **Param** and indexing type family $\mathbf{Idx}^p$consist of (1) a gradual type family $\mathbf{Acc}^p$ over p : **Param**, (2) a self-precise action $\otimes^p : \mathbf{Idx}^p \times \mathbf{Acc}^p \to \mathbf{Acc}^p$, (3) a distinguished self-precise family of terms $\mathbb{1}^p : \mathbf{Acc}^p$, satisfying the following precision inequations

$$i_1 \otimes i_3 \otimes i_2 \otimes acc \sqsubseteq i_1 \otimes i_2 \otimes acc \qquad i_1 \sqsubseteq i_3,\, i_2 \sqsubseteq i_3,\, (i_2 \otimes acc)^{\sqsubseteq},\, i_1, i_2, i_3 \in \mathbf{Idx}^p,\, acc \in \mathbf{Acc}^p \tag{1}$$

$$i_1 \otimes i_2 \otimes acc \sqsubseteq i_2 \otimes acc \qquad i_1 \sqsubseteq i_2,\, (i_2 \otimes acc)^{\sqsubseteq},\, i_1, i_2 \in \mathbf{Idx}^p,\, acc \in \mathbf{Acc}^p \tag{2}$$

$$i_1 \otimes acc \sqsubseteq i_2 \otimes i_1 \otimes acc \qquad i_1 \sqsubseteq i_2,\, (i_1 \otimes acc)^{\sqsubseteq},\, i_1, i_2 \in \mathbf{Idx}^p,\, acc \in \mathbf{Acc}^p \tag{3}$$

Since every component of an index accumulator are systematically parametrized by an instance p : **Param** of the telescope of parameters, we will omit it in the rest of the section. In particular, we identify $\mathbb{1}$: **Acc** with a single term and call it the empty accumulator. The inequalities in Definition 5.1 reflects properties featured by precision on the universe of types in [Maillard et al. 2022]: Equation (1) is the upper-decomposition property, Equation (2) reflects that downcasts gain precision, and Equation (3) reflects that upcasts lose precision. As a gradual family, **Acc** is also inhabited by the exceptional terms that play no role in the formalization of indexed inductive types.

We are specifically interested in index accumulators whose elements can be interpreted as sequences of indices. To that end, we observe that list of indices form the underlying type family of an indexed accumulator $\mathbf{Acc}_F$ that has an interesting property with respect to other index accumulators: it is the universal index accumulator with a unique map $\mathbb{r}$ to any other index accumulator. This can be phrased as a categorical property in the category of index accumulators, with monotone structure-preserving functions – meaning that empty is mapped to empty and actions to actions – as morphisms.

Lemma 5.2 (Initial Index Accumulator). *The initial index accumulator $\mathbf{Acc}_F$ exists and has* `list` $\boldsymbol{Idx}^p$ *as its underlying type. Moreover, precision for the free index accumulator admit an inductive presentation for self-precise* ***Param***, p : ***Param*** *and* $\boldsymbol{Idx}^p$ *presented in Figure 6.*

The unique morphism $\mathbb{r}$ from $\mathbf{Acc}_F$ to any index accumulator ***Acc*** *is the fold of the action starting from the empty accumulator:* $\mathbb{r}^p_{\mathbf{Acc}}\, acc \triangleq \boldsymbol{fold} \otimes_{\mathbf{Acc}} \mathbb{1}^p{}_{\mathbf{Acc}}\, acc.$

[4]The name Punk comes from our naming convention for the branch for the unknown term for an eliminator with motive P.

This explicit description of $\mathbf{Acc}_F$ provides in particular a function $\mathbf{hd} : \mathbf{Acc}_F \to \text{option}\, \mathbf{Idx}^p$ returning the head of a non-empty list. We can now turn to index accumulators whose elements can be interpreted as list of indices.

Definition 5.3 (Listable Index Accumulator). An index accumulator **Acc** is *listable* if the unique map $\mathbb{r} : \mathbf{Acc}_F \to \mathbf{Acc}$ has a left inverse $\mathbb{s} : \mathbf{Acc} \to \mathbf{Acc}_F$ that preserves the empty element and partially preserve the action

$$\mathbb{s}^p\, \mathbb{1} \rightsquigarrow^* [] \tag{4}$$

$$\mathbf{hd}\,(\mathbb{s}\,(\mathrm{i} \otimes acc)) \rightsquigarrow^* \text{some}\, i \quad \forall i : \mathbf{Idx},\ acc : \mathbf{Acc} \tag{5}$$

A listable index accumulator is a retract of the initial index accumulator $\mathbf{Acc}_F$ with retraction $\mathbb{r}$ and section $\mathbb{s}$. Equation (4) is required for Punk to be a conservative extension of CIC by ensuring that uncast terms are well-typed. Equation (5) is required for subject reduction by ensuring that casting a constructor (Rule G𝕀-Ctor-Cast) has the correct type. Note that $\mathbb{s}$ need *not* preserve the action. Indeed, $\mathbb{s}$ preserves the action if and only if it is an inverse of $\mathbb{r}$, so **Acc** and $\mathbf{Acc}_F$ are then isomorphic. To support other semantics (e.g., forgetful), we weaken this requirement.

A listable index accumulator **Acc** can be queried for the last element accumulated through a function $\mathbf{get} : \mathbf{Idx}^p \to \mathbf{Acc} \to \mathbf{Acc}$ defined as

$$\mathbf{get}\ i\ acc \triangleq \mathtt{match}\ \mathbf{hd}\,(\mathbb{s}\ acc)\ \mathtt{with}\ \text{none} \Rightarrow i \mid \text{some}\ i' \Rightarrow i'\ \ \text{end}$$

5.2 Examples of Index Accumulators

We describe three different instances of listable index accumulators (Figure 6) corresponding to the threes classes of cast reduction behavior identified in the introduction.

The Free accumulator $\mathbf{Acc}_F$*.* The initial index accumulator $\mathbf{Acc}_F$ faithfully accumulates every indices without losing any precision. By initiality, the precision on $\mathbf{Acc}_F$ consist of exactly those relations derivable from the inequations Eqs. (1) to (3) and the congruence rules given by self-precision of $\mathbb{1}^p$ and ⊗. It is listable with $\mathbb{s}_{\mathbf{Acc}_F} = \mathrm{id}_{\mathbf{Acc}_F}$.

The Meet accumulator $\mathbf{Acc}_M$*.* $\mathbf{Acc}_M$ follow the design of meet and combines intermediate casts using their meet with respect to precision, keeping at most two casts without any loss in precision. Meet accumulators provide a faithful optimization of the free semantics, meaning that they can error-approximate it using at most two casts.

The carrier of $\mathbf{Acc}_M$ has two inhabitants, zero and two carrying the corresponding number of elements of $\mathbf{Idx}^p$, and representing respectively the absence of casts and a meeting. zero is the empty accumulator and the action accumulates the meet of the indices in the second argument of two. Here, the meet operation ⊓ is inspired from the composition operator of Eremondi et al. [2022] that computes the greatest lower bound with respect to precision of any two terms. In the setting of GRIP, because not every term is self-precise, the specification of ⊓ needs to be relativized to self-precise types and terms

$$\text{For } T^{\sqsubseteq},\quad t_1^{\sqsubseteq_T} \wedge t_2^{\sqsubseteq_T} \quad\to\quad t_1 \sqcap t_2\ {}_T{\sqsubseteq}_T\ t_1 \quad\wedge\quad t_1 \sqcap t_2\ {}_T{\sqsubseteq}_T\ t_2.$$

The section $\mathbb{s}$ returns all the accumulated indices and ensures that the corresponding list of indices has length at most two. The precision on $\mathbf{Acc}_M$ stipulates that the two inhabitants are self-precise ($\mathbf{Acc}_M$-⊑-zero and $\mathbf{Acc}_M$-⊑-two).

The Forgetful accumulator $\mathbf{Acc}_U$*.* $\mathbf{Acc}_U$ discards any intermediate index and only preserves the last cast index, using the `option` type with inhabitants none and some as its underlying representation. The empty accumulator is none while the action always returns some of the newly

$\boxed{\text{Free accumulator } \mathbf{Acc}_F}$

$$\mathbf{Acc}_F^p \triangleq \texttt{list}\,\mathbf{Idx}^p \qquad i \otimes_F acc \triangleq i :: acc \qquad \mathbb{1}^p{}_F \triangleq []$$

$$\textsc{Acc}_F\text{-}\sqsubseteq\text{-nil}\ \frac{}{[] \sqsubseteq []} \qquad \textsc{Acc}_F\text{-}\sqsubseteq\text{-cons}\ \frac{i_1 \sqsubseteq i_2 \qquad acc_1 \sqsubseteq acc_2}{i_1 :: acc_1 \sqsubseteq i_2 :: acc_2} \qquad \textsc{Acc}_F\text{-}\sqsubseteq\text{-ret}\ \frac{i_1 \sqsubseteq i_3 \qquad i_2 \sqsubseteq i_3 \qquad acc^{\sqsubseteq}}{i_1 :: i_3 :: i_2 :: acc \sqsubseteq i_1 :: i_2 :: acc}$$

$$\textsc{Acc}_F\text{-}\sqsubseteq\text{-down}\ \frac{i_1 \sqsubseteq i_2 \qquad acc^{\sqsubseteq}}{i_1 :: i_2 :: acc \sqsubseteq i_2 :: acc} \qquad \textsc{Acc}_F\text{-}\sqsubseteq\text{-up}\ \frac{i_1 \sqsubseteq i_2 \qquad acc^{\sqsubseteq}}{i_1 :: acc \sqsubseteq i_2 :: i_1 :: acc} \qquad \textsc{Acc}_F\text{-}\sqsubseteq\text{-trans}\ \frac{acc_1 \sqsubseteq acc_2 \qquad acc_2 \sqsubseteq acc_3}{acc_1 \sqsubseteq acc_3}$$

$\boxed{\text{Meet accumulator } \mathbf{Acc}_M}$

$$\mathbf{Acc}_M^p \triangleq \mathsf{zero} \mid \mathsf{two}\,\mathbf{Idx}_M^p\,\mathbf{Idx}_M^p \qquad \mathbb{1}_M^p = \mathsf{zero}$$

$$\begin{aligned} i \otimes_M \mathsf{zero} &= \mathsf{two}\ i\ i \\ i \otimes_M (\mathsf{two}\ i'\ i'') &= \mathsf{two}\ i\ (i \sqcap i'') \end{aligned} \qquad \begin{aligned} \mathbb{s}_M^p\ \mathsf{zero} &= [] \\ \mathbb{s}_M^p\ (\mathsf{two}\ i\ i') &= [i, i'] \end{aligned}$$

$$\textsc{Acc}_M\text{-}\sqsubseteq\text{-zero}\quad \mathsf{zero} \sqsubseteq \mathsf{zero} \qquad \textsc{Acc}_M\text{-}\sqsubseteq\text{-two}\quad \mathsf{two}\ i_1\ i_2 \sqsubseteq \mathsf{two}\ i_1'\ i_2' \rightsquigarrow i_1 \sqsubseteq i_1' \land i_2 \sqsubseteq i_2'$$

$\boxed{\text{Forgetful accumulator } \mathbf{Acc}_U}$

$$\mathbf{Acc}_U^p \triangleq \mathsf{option}\,\mathbf{Idx}^p \qquad \mathbb{1}_U^p = \mathsf{none} \qquad i \otimes_U acc = \mathsf{some}\ i \qquad \begin{aligned} \mathbb{s}_U^p\ \mathsf{none} &= [] \\ \mathbb{s}_U^p\ (\mathsf{some}\ i) &= [i] \end{aligned}$$

$$\textsc{Acc}_U\text{-}\sqsubseteq\text{-none}\quad \mathsf{none} \sqsubseteq \mathsf{none} \qquad \textsc{Acc}_U\text{-}\sqsubseteq\text{-some}\quad \mathsf{some}\ i \sqsubseteq \mathsf{some}\ i' \rightsquigarrow i \sqsubseteq i'$$

Fig. 6. Instances of index accumulators

accumulated index, irrespective of the previous state of the accumulator. The section $\mathbb{s}_{\mathbf{Acc}_U}$ maps none to the empty list and some to a singleton list. The precision only needs to make none and some self-precise ($\textsc{Acc}_U$-$\sqsubseteq$-none and $\textsc{Acc}_U$-$\sqsubseteq$-some).

The forgetful index accumulator serves as a space optimization of the free index accumulator, although a really imprecise one. Since intermediate casts are ignored, computations will succeed as long as the most external cast is valid, allowing a program to recover from invalid states.

5.3 Static and Dynamic Semantics

In the subsequent sections, we detail the semantics of Punk for a fixed inductive family $\mathbb{I}$ with respect to a chosen listable index accumulator **Acc**. The typing and reduction rules for Punk, presented in Figure 7, are derived from the rules for indexed inductive types in CIC with a few adjustments to account for index accumulators. We reuse the metafunctions from the static semantics of indexed inductive types (Figure 3).

Typing. Constructors (G$\mathbb{I}$-Ctor) now hold additionally an index accumulator *acc* of type **Acc** and their indices are computed using **get** on said accumulator using the original index of the constructor

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

$$\mathbf{Head} ::= \ldots \mid \mathbb{I} \qquad \mathbf{Whnf}_{\square} ::= \ldots \mid \mathbb{I}\, p\, i \qquad \mathsf{head}\,(\mathbb{I}\, p\, i) := \mathbb{I}$$

$\boxed{\Gamma \vdash t : T}$

$$\textsc{G}\mathbb{I}\textsc{-Ty}\ \frac{\Gamma \vdash p : \mathbf{Param} \qquad \Gamma \vdash i : \mathbf{Idx}^{p}}{\Gamma \vdash \mathbb{I}\, p\, i : \square_{\ell}} \qquad \textsc{G}\mathbb{I}\textsc{-Err}\ \frac{\Gamma \vdash p : \mathbf{Param} \qquad \Gamma \vdash i : \mathbf{Idx}^{p}}{\Gamma \vdash \mathsf{err}_{\mathbb{I}\, p\, i} : \mathbb{I}\, p\, i} \qquad \textsc{G}\mathbb{I}\textsc{-Unk}\ \frac{\Gamma \vdash p : \mathbf{Param} \qquad \Gamma \vdash i : \mathbf{Idx}^{p}}{\Gamma \vdash ?_{\mathbb{I}\, p\, i} : \mathbb{I}\, p\, i}$$

$$\textsc{G}\mathbb{I}\textsc{-Ctor}\ \frac{\begin{array}{c}\Gamma \vdash p : \mathbf{Param} \qquad \Gamma \vdash a : \mathbf{Arg}^{p}_{\mathbb{c}} \\ \Gamma \vdash acc : \mathbf{Acc}^{p} \qquad \forall\, j < \#\mathbf{indargs}_{\mathbb{c}},\ \Gamma \vdash r_j : \Pi(x_j : \mathbf{IndArg}^{p}_{\mathbb{c},j}\ a), \mathbb{I}\, p\, (\mathbf{arg\_idx}^{p}_{\mathbb{c},j}\ a\, x_j)\end{array}}{\Gamma \vdash \mathbb{c}^{p}\, acc\, a\, \overrightarrow{r_j} : \mathbb{I}\, p\, (\mathbf{get}^{p}\ (\mathbf{ctr\_idx}^{p}_{\mathbb{c}}\ a)\ acc)}$$

$$\textsc{G}\mathbb{I}\textsc{-Catch}\ \frac{\begin{array}{c}\Gamma \vdash p : \mathbf{Param} \qquad \Gamma \vdash M : \Pi(i : \mathbf{Idx}^{p}), \mathbb{I}\, p\, i \to \square_{\ell'} \qquad \Gamma \vdash i : \mathbf{Idx}^{p} \qquad \Gamma \vdash t : \mathbb{I}\, p\, i \\ \Gamma \vdash h_{\mathsf{err}} : \Pi(i : \mathbf{Idx}^{p}), M\, i\, \mathsf{err}_{\mathbb{I}\, p\, i} \qquad \Gamma \vdash h_{\mathsf{unk}} : \Pi(i : \mathbf{Idx}^{p}), M\, i\, ?_{\mathbb{I}\, p\, i} \qquad \forall\, k < \#\mathbf{ctors}, \Gamma \vdash h_{\mathbb{c}_k} : \mathsf{Branch}^{p}_{\mathbb{c}_k}\ M\end{array}}{\Gamma \vdash \mathsf{catch}^{p}\ M\ \overrightarrow{h_{\mathbb{c}_k}}\ h_{\mathsf{err}}\ h_{\mathsf{unk}}\ i\, t : M\, i\, t}$$

$\boxed{t \rightsquigarrow t}$

$\textsc{G}\mathbb{I}\textsc{-Catch-Ctor}$: $\mathsf{catch}^{p}\ M\ \overrightarrow{h_{\mathbb{c}_k}}\ h_{\mathsf{err}}\ h_{\mathsf{unk}}\ i\ (\mathbb{c}^{p}\, acc\, a\, \overrightarrow{r_j}) \rightsquigarrow \langle M_n \Leftarrow \ldots M_0 \rangle\ h_{\mathbb{c}}\ a\, \overrightarrow{r_j}\ \overrightarrow{t_j}$

with $[i, \ldots, i_k, \ldots\ i_1] \triangleq \mathbb{s}^{p}\ acc \qquad acc_{\mathsf{k}} \triangleq \mathbb{r}^{p}\ [i_k, \ldots, i_1] \qquad M_k \triangleq M\, i_k\ (\mathbb{c}^{p}\, acc_{\mathsf{k}}\, a\, \overrightarrow{r_j}) \qquad M_n \triangleq M\, i\ (\mathbb{c}^{p}\, acc\, a\, \overrightarrow{r_j})$

$M_0 \triangleq M\ (\mathbf{ctr\_idx}^{p}_{\mathbb{c}}\ a)\ (\mathbb{c}^{p}\, \mathbb{1}^{p}\ a\, \overrightarrow{r_j}) \qquad t_j \triangleq \lambda(x_j : \mathbf{IndArg}^{p}_{\mathbb{c},j}\ a), \mathsf{catch}^{p}\ M\ \overrightarrow{h_{\mathbb{c}_k}}\ h_{\mathsf{err}}\ h_{\mathsf{unk}}\ (\mathbf{arg\_idx}^{p}_{\mathbb{c},j}\ a\, x_j)\ (r_j\, x_j)$

$$\textsc{G}\mathbb{I}\textsc{-Catch-Err}\text{: } \mathsf{catch}^{p}\ M\ \overrightarrow{h_{\mathbb{c}_k}}\ h_{\mathsf{err}}\ h_{\mathsf{unk}}\ i\ \mathsf{err}_{\mathbb{I}\, p\, i} \rightsquigarrow h_{\mathsf{err}}\ i$$

$$\textsc{G}\mathbb{I}\textsc{-Catch-Unk}\text{: } \mathsf{catch}^{p}\ M\ \overrightarrow{h_{\mathbb{c}_k}}\ h_{\mathsf{err}}\ h_{\mathsf{unk}}\ i\, ?_{\mathbb{I}\, p\, i} \rightsquigarrow h_{\mathsf{unk}}\ i$$

$\textsc{G}\mathbb{I}\textsc{-Ctor-Cast}$: $\langle \mathbb{I}\, p'\ i' \Leftarrow \mathbb{I}\, p\, i \rangle\ \mathbb{c}^{p}\, acc\, a\, \overrightarrow{r_j} \rightsquigarrow \mathbb{c}^{p'}\ (i' \otimes^{p'}\ acc')\ a'\ \overrightarrow{r'_j}$

with $\quad a' \triangleq \langle \mathbf{Arg}^{p'}_{\mathbb{c}} \Leftarrow \mathbf{Arg}^{p}_{\mathbb{c}} \rangle\ a \qquad acc' \triangleq \langle \mathbf{Acc}^{p'} \Leftarrow \mathbf{Acc}^{p} \rangle\ (i \otimes^{p} acc)$

$$r'_j \triangleq \langle \Pi(x'_j : \mathbf{IndArg}^{p'}_{\mathbb{c},j}\ a'), \mathbb{I}\, p'\ (\mathbf{arg\_idx}^{p'}_{\mathbb{c},j}\ a'\ x'_j) \Leftarrow \Pi(x_j : \mathbf{IndArg}^{p}_{\mathbb{c},j}\ a), \mathbb{I}\, p\, (\mathbf{arg\_idx}^{p}_{\mathbb{c},j}\ a\, x_j) \rangle\ r_j$$

$\textsc{G}\mathbb{I}\textsc{-Err-Cast}$: $\langle \mathbb{I}\, p'\ i' \Leftarrow \mathbb{I}\, p\, i \rangle\ \mathsf{err}_{\mathbb{I}\, p\, i} \rightsquigarrow \mathsf{err}_{\mathbb{I}\, p'\ i'}$ $\qquad \textsc{G}\mathbb{I}\textsc{-Unk-Cast}$: $\langle \mathbb{I}\, p'\ i' \Leftarrow \mathbb{I}\, p\, i \rangle\ ?_{\mathbb{I}\, p\, i} \rightsquigarrow ?_{\mathbb{I}\, p'\ i'}$

$\textsc{G}\mathbb{I}\textsc{-Dec-Up}$: $\langle ?_{\square} \Leftarrow \mathbb{I}\, p\, i \rangle\ t \rightsquigarrow \langle ?_{\square} \Leftarrow \mathbb{I}\, ?_{\mathbf{Param}}\ i' \rangle\ \langle \mathbb{I}\, ?_{\mathbf{Param}}\ i' \Leftarrow \mathbb{I}\, p\, i \rangle\ t \qquad$ with $\quad i' \triangleq \langle \mathbf{Idx}^{?_{\mathbf{Param}}} \Leftarrow \mathbf{Idx}^{p} \rangle\ i$

$\textsc{G}\mathbb{I}\textsc{-Dec-Down}$: $\langle \mathbb{I}\, p\, i \Leftarrow ?_{\square} \rangle\ t \rightsquigarrow \langle \mathbb{I}\, p\, i \Leftarrow \mathbb{I}\, ?_{\mathbf{Param}}\ i' \rangle\ \langle \mathbb{I}\, ?_{\mathbf{Param}}\ i' \Leftarrow ?_{\square} \rangle\ t$

$\textsc{G}\mathbb{I}\textsc{-Head-Err}$: $\langle X \Leftarrow \mathbb{I}\, p\, i \rangle\ t \rightsquigarrow \mathsf{err}_X \qquad$ when $X \in \mathbf{Whnf}_{\square}$ and $\mathbb{I} \neq \mathsf{head}\, X$

Evaluation contexts

$C ::= \ldots \mid \langle C \Leftarrow A \rangle\ t \mid \langle B \Leftarrow C \rangle\ t \mid \langle B \Leftarrow A \rangle\ C \mid \mathsf{catch}^{p}\ M\ \overrightarrow{h_{\mathbb{c}_k}}\ h_{\mathsf{err}}\ h_{\mathsf{unk}}\ i\, C$

$$\textsc{GRed-Cong}\ \frac{t \rightsquigarrow t'}{C[t] \rightsquigarrow C[t']}$$

$\boxed{\mathtt{canonical}\ t}$

$$\frac{}{\mathtt{canonical}\ \mathsf{err}_{\mathbb{I}\, p\, i}} \qquad \frac{}{\mathtt{canonical}\ ?_{\mathbb{I}\, p\, i}} \qquad \frac{}{\mathtt{canonical}\ \mathbb{c}^{p}\, acc\, a\, \overrightarrow{r_j}}$$

Fig. 7. Typing and reduction rules for Punk.

as default value. As with any gradual type, the exceptional terms `err` (G𝕀-Err) and ? (G𝕀-Unk) inhabit the inductive type for any valid instance of the parameters and indices. The eliminator `catch` (G𝕀-Catch) extends `elim` from CIC with two new branches, one for each exceptional term. Finally, the typing of indexed inductive types (G𝕀-Ty) remains unchanged.

Reduction. The canonical forms for an inductive family consist of constructors as in CIC, as well as both exceptional terms $?_{\mathbb{I}\,p\,i}$ and $\mathtt{err}_{\mathbb{I}\,p\,i}$. The crux of the dynamic semantics is concentrated in two reduction rules that describe how index accumulators are used to accumulate the indices of casts on constructors and how they are used when eliminating a constructor.

Casting a constructor (G𝕀-Ctor-Cast) to the same inductive type, with possibly different parameters and indices, is performed as follows: First, the index accumulator acc, the non-inductive arguments a, and the inductive arguments $\overrightarrow{r_j}$ are cast so they now depend on the target parameter p'. Then, we add the target index i' to the previously cast index accumulator using $\otimes$. Since parameters are uniform over an indexed inductive definition, they need to be processed differently from indices. Treating parameters as indices would lead to a less useful elimination principle where the motive would have to abstract over parameters as well as indices. Casts propagate exceptional terms (G𝕀-Err-Cast and G𝕀-Unk-Cast) adapting the indices and parameters on the fly. Casting a term of an inductive type to and from the unknown type $?_\square$ (G𝕀-Dec-Up and G𝕀-Dec-Down) is decomposed into two casts, going through $\mathbb{I}\,?_{\mathbf{Param}}\ (\langle \mathbf{Idx}^{?_\square} \Leftarrow \mathbf{Idx}^{p}\rangle\ i)$, the most general type with $\mathbb{I}$ as the head constructor at the "same" index (cast to the correct type). Although going through the most general index $?_{\mathbf{Idx}^{?\mathrm{Param}}}$ would also work we prefer our system to be as precise as possible, while preserving graduality (Definition 3.3), and not lose precision if it is not required. Finally, casts from an indexed inductive type to a different indexed inductive type (e.g., $\langle a =_A b \Leftarrow \mathbb{V}\,A\,n\rangle\ v$), a Π type, or a universe always fail (G𝕀-Head-Err). Following GRIP, we present this reduction rule using an auxilliary function head computing the head type former of a type in weak head normal form (e.g., $\mathsf{head}\,(\mathbb{V}\,A\,n) = \mathbb{V}$). This complete the description of reduction on casts, and we now focus on the reduction on `catch`.

Elimination of constructors using `catch` (G𝕀-Catch-Ctor) is more involved since we need to describe how to handle the casts accumulated in the index accumulator. At a high level, the rule propagates casts to the outside of the `catch` using the list of indices to construct the target of each cast and applies the corresponding branch as if the accumulator was empty. To facilitate the comprehension of the rule we present a detailed step-by-step description of how this reduction rule works:

(1) The branch corresponding to the constructor is applied ignoring the accumulator.
(2) The index accumulator acc is converted into a list of indices $\mathbb{s}\ acc$ from which we compute a sequence $[acc_{\mathsf{n}}, \ldots, acc_{1}]$ of partial accumulators using $\otimes$ starting from $\mathbb{1}$; in particular, $acc_{\mathsf{n}} \triangleq \mathbb{r}^{p}\ (\mathbb{s}\ acc)$.
(3) The intermediate cast types are then defined as $M_k \triangleq M\ (\mathbf{get}^{p}\ (\mathbf{ctr\_idx}^{p}_{\mathbb{c}}\ a)\ acc_{\mathsf{k}})\ (\mathbb{c}^{p}\, acc_{\mathsf{k}}\ a\,\overrightarrow{r_j})$.
(4) Finally, the right hand side of the reduction rule consist of n casts on the result of applying the correct branch (step 1), passing through each intermediate target M_k in order, and ending with the motive M applied to the eliminated term $\mathbb{c}^{p}\, acc\ a\,\overline{r_j}$ and its index i.

When the index accumulator is empty, this rule is equivalent to the reduction for `elim` on constructors (𝕀-Elim-Ctor in Figure 5).

The interaction of `catch` on exceptional terms uses the corresponding premises (respectively h_{err} in G𝕀-Catch-Err and h_{unk} for G𝕀-Catch-Unk). Finally, we have some congruence rules in order for terms eliminated using `catch` reduce until canonical forms.

$$\textsc{$\mathbb{I}$-$\sqsubseteq$-Cong-Ty}\quad \mathbb{I}\,p\,i \sqsubseteq_\ell \mathbb{I}\,p'\,i' \rightsquigarrow p \;{}_{\mathbf{Param}}{\sqsubseteq}_{\mathbf{Param}'}\; p' \wedge i \;{}_{\mathbf{Idx}^{p}}{\sqsubseteq}_{\mathbf{Idx}^{p'}}\; i'$$

$$\texttt{err}\text{-}\mathbb{I}\text{-}\sqsubseteq\text{-}\textsc{Ty}\quad \frac{\Gamma \vdash w : \mathbb{I}\,p\,i^{\sqsubseteq_\ell}}{\Gamma \vdash \texttt{err-min-ty}\; w : \texttt{err}_\square \sqsubseteq_\ell \mathbb{I}\,p\,i}$$

$$?\text{-}\mathbb{I}\text{-}\sqsubseteq\text{-}\textsc{Ty}\quad \frac{\Gamma \vdash w : \mathbb{I}\,p\,i^{\sqsubseteq_\ell}}{\Gamma \vdash \texttt{?-max-ty}\; w : \mathbb{I}\,p\,i \sqsubseteq_\ell ?_\square}$$

$$\textsc{NoConf-$\mathbb{I}$-Up-Ty}\quad \frac{X \in \mathbf{Whnf}_\square \qquad \mathsf{head}\, X \neq \mathbb{I}, ?_\square}{\mathbb{I}\,p\,i \sqsubseteq_\ell X \rightsquigarrow \bot}$$

$$\textsc{NoConf-$\mathbb{I}$-Down-Ty}\quad \frac{X \in \mathbf{Whnf}_\square \qquad \mathsf{head}\, X \neq \mathbb{I}, \texttt{err}_\square}{X \sqsubseteq_\ell \mathbb{I}\,p\,i \rightsquigarrow \bot}$$

$$\texttt{err}\text{-}\mathbb{I}\text{-}\sqsubseteq\quad \frac{\Gamma \vdash w_{\mathbb{I}} : \mathbb{I}\,p\,i^{\sqsubseteq_\ell} \qquad \Gamma \vdash w'_{\mathbb{I}} : \mathbb{I}\,p'\,i'^{\sqsubseteq_\ell} \qquad \Gamma \vdash w_x : x^{\sqsubseteq_{\mathbb{I}\,p'\,i'}}}{\Gamma \vdash \texttt{err -min}\; w_{\mathbb{I}}\; w'_{\mathbb{I}}\; w_x : \texttt{err}_{\mathbb{I}\,p\,i} \;{}_{\mathbb{I}\,p\,i}{\sqsubseteq}_{\mathbb{I}\,p'\,i'}\; x}$$

$$\begin{aligned}\mathbb{I}\text{-}\sqsubseteq\text{-}\mathbb{c}\quad & \mathbb{c}^{p}\, acc\; a\, \overline{r_j} \;{}_{\mathbb{I}\,p\,i}{\sqsubseteq}_{\mathbb{I}\,p'\,i'}\; \mathbb{c}^{p'}\, acc'\; a'\, \overline{r'_j} \rightsquigarrow i \otimes acc \;{}_{\mathbf{Acc}^{p}}{\sqsubseteq}_{\mathbf{Acc}^{p'}}\; i' \otimes acc' \wedge a \;{}_{\mathbf{Arg}^{p}_{\mathbb{c}}}{\sqsubseteq}_{\mathbf{Arg}^{p'}_{\mathbb{c}}}\; a' \\ & \wedge \bigwedge_j (r_j \;{}_{\Pi(x_j:\mathbf{IndArg}^{p}_{\mathbb{c},j}\, a), \mathbb{I}\,p\,(\mathbf{arg\_idx}^{p}_{\mathbb{c},j}\, a\, x_j)}{\sqsubseteq}_{\Pi(x'_j:\mathbf{IndArg}^{p'}_{\mathbb{c},j}\, a'), \mathbb{I}\,p'\,(\mathbf{arg\_idx}^{p'}_{\mathbb{c},j}\, a'\, x'_j)}\; r'_j)\end{aligned}$$

$$?\text{-}\mathbb{I}\text{-}\sqsubseteq\quad \frac{\Gamma \vdash w_{\mathbb{I}} : \mathbb{I}\,p\,i^{\sqsubseteq_\ell} \qquad \Gamma \vdash w'_{\mathbb{I}} : \mathbb{I}\,p'\,i'^{\sqsubseteq_\ell} \qquad \Gamma \vdash w_x : x^{\sqsubseteq_{\mathbb{I}\,p\,i}}}{\Gamma \vdash \texttt{?-max}\; w_{\mathbb{I}}\; w'_{\mathbb{I}}\; w_x : x \;{}_{\mathbb{I}\,p\,i}{\sqsubseteq}_{\mathbb{I}\,p'\,i'}\; ?_{\mathbb{I}\,p'\,i'}}$$

$$\textsc{NoConf-$\mathbb{c}$}\quad \frac{j \neq k}{\mathbb{c}^{p}_j\, acc\; a\, \overrightarrow{r_j} \;{}_{\mathbb{I}\,p\,i}{\sqsubseteq}_{\mathbb{I}\,p'\,i'}\; \mathbb{c}^{p'}_k\, acc'\; a'\, \overrightarrow{r'_j} \rightsquigarrow \bot}$$

Fig. 8. Precision for indexed inductive types.

5.4 Precision

Now, we turn our focus to the definition of precision for gradual indexed inductive types (Figure 8). Precision for telescopes is defined as pairwise precision. Indexed inductive types and their constructors are monotonous. Therefore, two indexed inductive types are related by precision exactly when they have the same head and their parameters and indices are also related ($\mathbb{I}$-$\sqsubseteq$-Cong-Ty). The exceptional types $\texttt{err}_\square$ and $?_\square$ are the bottom and top elements respectively for self-precise indexed inductive types ($\texttt{err}$-$\mathbb{I}$-$\sqsubseteq$-Ty and $?$-$\mathbb{I}$-$\sqsubseteq$-Ty). Indexed inductive types are never related to types with a different head constructor (NoConf-$\mathbb{I}$-Up-Ty and NoConf-$\mathbb{I}$-Down-Ty). Two constructors are related by precision exactly when they are the same constructor and their arguments are pairwise precise ($\mathbb{I}$-$\sqsubseteq$-$\mathbb{c}$). As expected, the exceptional terms $\texttt{err}$ and $?$ serve as the bottom and top elements respectively for self-precise terms ($\texttt{err}$-$\mathbb{I}$-$\sqsubseteq$ and $?$-$\mathbb{I}$-$\sqsubseteq$) and self-precise parameters and indices. Also, for each pair of different constructors $\mathbb{c}_j$ and $\mathbb{c}_k$ where $j \neq k$ there is a no confusion rule (NoConf-$\mathbb{c}$) stating that different constructors are never related by precision.

6 Metatheory of Punk

This section establishes the main metatheoretical properties of the Punk framework: subject reduction, conservativity over CIC (§6.1) and graduality (§6.2). These properties hold for any choice of a listable index accumulator, hence for all the gradual indexed inductive types defined using the instances of §5.2. Section §6.3 then shows that these different instances of index accumulator exhibit the three possible behavior of eliminators with respect to precision.

6.1 Basic Metatheory of the Punk Framework

As an extension of GRIP [Maillard et al. 2022] with comprehensive support for indexed inductive types, Punk preserves two important properties as a gradual dependently-typed programming language: *subject reduction* and *conservativity* with respect to CIC. Throughout the section, we focus on the fragment specific to indexed inductive types and rely on the established metatheory of GRIP when needed. We center the discussion towards the properties of constructors, sidelining exceptional terms, as focusing on these properties highlights the nuanced insights that have guided the design choices of Punk's formalization (§5). For brevity, this section presents definitions and theorems in an empty context, focusing on a generic inductive family $\mathbb{I} : \Pi(p : \mathbf{Param}), \mathbf{Idx}^{p} \rightarrow \square_{\ell}$, for any choice of a listable index accumulator $\mathbf{Acc}$.

Conversion for inductive families. Conversion is defined in terms of reduction and syntactic comparison of terms up to extensionality principles—proof irrelevance for strict propositions and η-rule for functions. Two terms t and u are convertible, noted $t \equiv u$, if both eventually reduce to two terms that are extensionally equal, noted $t =_{\eta} u$.

The notion of conversion is crucial in a dependently typed setting to support computation at the level of types.

$$\textsc{Conv}\quad \frac{\Gamma \vdash t : T' \qquad \Gamma \vdash T : \square_{\ell} \qquad \Gamma \vdash T' \equiv T : \square_{\ell}}{\Gamma \vdash t : T}$$

Conversion and the conversion typing rule (Conv) play an important role in the proof of subject reduction. In particular the proof depends on the injectivity of the type constructor for indexed inductive types. With the definition of conversion based on reduction, injectivity of type constructors is easy to prove because there is no reduction rules applying to them and extensional equality is straightforwardly injective on type constructors.

Note that the definition of conversion using reduction however has the drawback that it does not immediately form an equivalence relation, and in particular transitivity is not for free. This property has been formally proven by Sozeau et al. [2019] for the type theory of the Coq proof assistant. The crux of the proof lies in the proof of confluence (Church-Rosser) of reduction together with a simulation lemma saying that extensional equality commutes with reduction. In Punk, the reduction rules have been extended to guarantee that the technique used for the proof of confluence remains valid. The proof of confluence makes use of the auxiliary notion of one-step parallel reduction, noted $t \Rrightarrow u$, which allows all available reductions to be performed in parallel in the same step. One-step parallel reduction satisfies the triangle lemma saying that for any term t, there exists an optimally reduced term $\rho(t)$ (that performs all possible reduction in parallel) such that $t \Rrightarrow \rho(t)$ and for any $t \Rrightarrow u$, $u \Rrightarrow \rho(t)$. Confluence is then a direct consequence of this triangle lemma and the fact that parallel reduction entails reduction.

Subject reduction. We first prove that typing is preserved by reduction. The main two cases concern cast reduction on a constructor and elimination via `catch`.

Lemma 6.1 (Subject reduction for casts on constructors). *Consider two indices i_1, i_2, a listable index accumulator acc, and arguments a and $\overrightarrow{r_j}$. If $\vdash \langle \mathbb{I}\, p_2\, i_2 \Leftarrow \mathbb{I}\, p_1\, i_1 \rangle\, \mathbb{c}^{p_1}\, acc\, a\, \overrightarrow{r_j} : \mathbb{I}\, p_2\, i_2$, then*

$$\vdash \mathbb{c}^{p_2}\, (i_2 \otimes acc')\; a'\; (\langle T_j \Leftarrow C_j \rangle\, \overrightarrow{r_j}) : \mathbb{I}\, p_2\, i_2$$

where $\forall\, j.\, T_j \triangleq \Pi(x_j : \boldsymbol{IndArg}^{p_2}_{\mathbb{c},j}\, a'), \mathbb{I}\, p\, (\boldsymbol{arg\_idx}^{p_2}_{\mathbb{c},j}\, a'\, x_j)$, $a' \triangleq \langle \boldsymbol{Arg}^{p_2}_{\mathbb{c}} \Leftarrow B \rangle\, a$, and $acc' \triangleq \langle \boldsymbol{Acc}^{p_2} \Leftarrow A \rangle\, (i_1 \otimes acc)$, for some terms A, B, and $\forall\, j.\, C_j$.

Proof. By repeated inversion on the typing derivation of $\langle \mathbb{I}\, p_2\, i_2 \Leftarrow \mathbb{I}\, p_1\, i_1 \rangle\, \mathbb{c}^{p_1}\, acc\, a\, \overrightarrow{r_j}$, we know that $i_1 : \mathbf{Idx}^{p_1}$ and $i_2 : \mathbf{Idx}^{p_2}$, and $\mathbb{c}^{p_1}\, acc\, a\, \overrightarrow{r_j} : \mathbb{I}\, p_1\, i_1$. First, we have to show

$$\mathbb{I}\, p_2\, \mathbf{get}^{p_2}\, (\mathbf{ctr\_idx}^{p_2}_{\mathbb{c}}\, a')\, (i_2 \otimes acc') \equiv \mathbb{I}\, p_2\, i_2.$$

This equation is given by the injectivity of the type constructor $\mathbb{I}$, the definition of **get**, and the partial preservation of the action by $\mathbb{s}$ (Equation (5)). Then, by inversion on the typing derivation of $\mathbb{c}^{p_1}\, acc\, a\, \overrightarrow{r_j} : \mathbb{I}\, p_1\, i_1$, and the rules for conversion (Conv) and typing of constructors (G$\mathbb{I}$-Ctor), we have $acc' : A$, $a : B$, and $\forall\, j.\, r_j : C_j$. Finally, by typing of constructors (G$\mathbb{I}$-Ctor), we have to show

$$i_2 \otimes acc' : \mathbf{Acc}^{p_2} \qquad\qquad a' : \mathbf{Arg}_{\mathbb{c}}^{p_2} \qquad\qquad \forall\, j.\, \langle T_j \Leftarrow C_j \rangle\, \overrightarrow{r_j} : T_j.$$

These typing judgments follow directly from the typing rule of casts (§2.1). □

Lemma 6.2 (Subject reduction for catch on constructors). *Consider a parameter p; a motive M; branches $\overrightarrow{P\mathbb{c}_k}$, h_{err}, and h_{unk}; an index i; a listable index accumulator acc; and arguments a and $\overrightarrow{r_j}$. If $\vdash \mathsf{catch}^p\, M\, \overrightarrow{h_{\mathbb{c}}}\, h_{\mathsf{err}}\, h_{\mathsf{unk}}\, i\, (\mathbb{c}^p\, acc\, a\, \overrightarrow{r_j}) : M\, i\, (\mathbb{c}^p\, acc\, a\, \overrightarrow{r_j})$, then*

$$\vdash \langle M\, i\, (\mathbb{c}^p\, acc\, a\, \overrightarrow{r_j}) \Leftarrow \ldots M\, (\boldsymbol{ctr\_idx}_{\mathbb{c}}^p\, a)\, (\mathbb{c}^p\, \mathbb{1}^p\, a\, \overrightarrow{r_j}) \rangle\, h_{\mathbb{c}}\, a\, \overrightarrow{r_j}\, \overrightarrow{t_j} : M\, i\, (\mathbb{c}^p\, acc\, a\, \overrightarrow{r_j}).$$

Proof. By inversion on the typing rule of catch (G$\mathbb{I}$-Catch) we know,

$$p : \mathbf{Param} \qquad h_{\mathbb{c}} : \mathsf{Branch}_{\mathbb{c}}^p\, M \qquad M : \Pi(i' : \mathbf{Idx}^p).\, \mathbb{I}\, p\, i' \to \square \qquad \mathbb{c}^p\, acc\, a\, \overrightarrow{r_j} : \mathbb{I}\, p\, i.$$

Then, we have to show that (a) the outermost target is well-typed and that (b) the method application is well-typed to the source type. (a) follows directly from the type of M, i, and $\mathbb{c}^p\, acc\, a\, \overrightarrow{r_j}$; (b) by the conversion rule (Conv) and the typing rule for constructors (G$\mathbb{I}$-Ctor), we have to show, between some typing premises, that $\mathbb{I}\, p\, (\mathbf{get}^p\, (\mathbf{ctr\_idx}_{\mathbb{c}}^p\, a)\, \mathbb{1}^p) \equiv \mathbb{I}\, p\, (\mathbf{ctr\_idx}_{\mathbb{c}}^p\, a)$, which is given directly by the injectivity of the type constructor $\mathbb{I}$, the definition of **get**, and the preservation of the empty element by $\mathbb{s}$ (Eq. (4)). The rest of the premises are given by the inversion of the hypothesis. □

Theorem 6.3 (Subject reduction for Punk). *If $\vdash T : \square_{\ell'}$, $\vdash t : T$ and $t \rightsquigarrow t'$ for some t', then $\vdash t' : T'$ for some $\vdash T' : \square_{\ell'}$ such that $\vdash T \equiv T'$.*

Proof. The proof proceeds by induction on $t \rightsquigarrow t'$. The cases for casts (G$\mathbb{I}$-Ctor-Cast) and catch (G$\mathbb{I}$-Catch-Ctor) on constructors follow directly from Lemmas 6.1 and 6.2. Cases for casts and catch on exceptional terms follow directly from the typing rules. The rest of the cases are given by the induction hypothesis, the typing rules, and the subject reduction of GRIP. □

Conservativity with respect to CIC. Since Punk extends the syntax of constructors and even uses a different eliminator than the static language presented in §4, we cannot reuse CastCIC's *static term* definition (i.e., a term of CastCIC that it is also a term of CIC) and need to give a specific translation (noted $[\![\_]\!]$) from CIC to Punk for indexed inductive types, constructors and applications of catch:

$$[\![\mathbb{I}\, p\, i]\!] \triangleq \mathbb{I}\, [\![p]\!]\, [\![i]\!] \qquad\qquad [\![\mathbb{c}^p\, a\, \overrightarrow{r}]\!] \triangleq \mathbb{c}^{[\![p]\!]}\, \mathbb{1}^{[\![p]\!]}\, [\![a]\!]\, \overrightarrow{[\![r]\!]}$$

$$[\![\mathsf{elim}^p\, M\, \overrightarrow{h_{\mathbb{c}}}\, i\, t]\!] \triangleq \mathsf{catch}^{[\![p]\!]}\, [\![M]\!]\, (\lambda i'.\, \mathsf{err}_{[\![M]\!]\, i'\, \mathsf{err}_{\mathbb{I}\, [\![p]\!]\, i'}})\, (\lambda i'.\, ?_{[\![M]\!]\, i'\, ?_{\mathbb{I}\, [\![p]\!]\, i'}})\, \overrightarrow{[\![h_{\mathbb{c}}]\!]}\, [\![i]\!]\, [\![t]\!]$$

Theorem 6.4 (Conservativity of Punk over CIC). *For any term t and type T of CIC, if $\vdash_{\mathsf{CIC}} t : T$ then $\vdash [\![t]\!] : [\![T]\!]$.*

Proof. By induction on the typing derivation of $\vdash_{\mathsf{CIC}} t : T$. The case for constructors follows from Equation (4). The case for catch is direct by the translation. □

6.2 Gradual Properties of Punk

An inductive family designed within the Punk framework adheres to all the characteristics of a gradual inductive family outlined in § 3.2, specifically *graduality*, as well as *index relevance* and *shape relevance*. Similarly to § 6.1, we focus on the fragment specific to indexed inductive types, particularly to the case of constructors. We state definitions and theorems in the empty context, in terms of a generic inductive family $\mathbb{I}$, and for any choice of index accumulator $\mathbf{Acc}$.

Graduality for Inductive Families. We now prove graduality for inductive families, i.e., inductive families are monotone with respect to precision. More precisely, we prove that up-casting and down-casting the parameters and indices of an indexed inductive type form an embedding-projection pair with respect to precision [New and Ahmed 2018], i.e., satisfy the three following properties: (1) casts have to be monotone; (2) casting to a less precise type and back is the identity up to equiprecision (unit of the Galois connection and retraction condition); (3) casting to a more precise type and back increases precision (counit of the galois connection). We state these properties for a fixed parameter p and only on constructors where the most important features of Punk are apparent.

Lemma 6.5 (Graduality for constructors). *Let $p : \mathbf{Param}$ be a self-precise parameter, $i_1, i_2 : \mathbf{Idx}^p$, $acc : \mathbf{Acc}^p$ a self-precise index accumulator, and self-precise arguments $a : \mathbf{Arg}^p_{\mathbb{c}}$ and $\overrightarrow{r_j}$, such that $\vdash \mathbb{c}^p\, acc\, a\, \overrightarrow{r_j} : \mathbb{I}\, p\, i_1$. Then, the following precision relations hold*

$$\begin{array}{l} i_1 \ {}_{\mathbf{Idx}^p}{\sqsubseteq}_{\mathbf{Idx}^p}\ i_2 \rightarrow \mathbb{c}^p\, acc\, a\, \overrightarrow{r_j} \sqsupseteq\sqsubseteq_{\mathbb{I}\,p\,i_1} \langle \mathbb{I}\, p\, i_1 \Leftarrow \mathbb{I}\, p\, i_2 \Leftarrow \mathbb{I}\, p\, i_1 \rangle\, \mathbb{c}^p\, acc\, a\, \overrightarrow{r_j} \\ i_2 \ {}_{\mathbf{Idx}^p}{\sqsubseteq}_{\mathbf{Idx}^p}\ i_1 \rightarrow \langle \mathbb{I}\, p\, i_1 \Leftarrow \mathbb{I}\, p\, i_2 \Leftarrow \mathbb{I}\, p\, i_1 \rangle\, \mathbb{c}^p\, acc\, a\, \overrightarrow{r_j}\ {}_{\mathbb{I}\,p\,i_1}{\sqsubseteq}_{\mathbb{I}\,p\,i_1}\ \mathbb{c}^p\, acc\, a\, \overrightarrow{r_j}. \end{array}$$

Proof. By reduction of cast on constructors (G$\mathbb{I}$-Ctor-Cast), and removing the identity casts that appears (which is valid up-to equiprecision), we have

$$\langle \mathbb{I}\, p\, i_1 \Leftarrow \mathbb{I}\, p\, i_2 \Leftarrow \mathbb{I}\, p\, i_1 \rangle\, \mathbb{c}^p\, acc\, a\, \overrightarrow{r_j} \rightsquigarrow \mathbb{c}^p (i_1 \otimes i_2 \otimes i_2 \otimes i_1 \otimes acc)\, a\, \overrightarrow{r_j}$$

By the definition of precision on constructors ($\mathbb{I}$-$\sqsubseteq$-$\mathbb{c}$), we have to show (omiting type annotation of precision for readability)

$$\begin{array}{lcl} i_1 \sqsubseteq i_2 & \rightarrow & i_1 \otimes acc \sqsubseteq i_1 \otimes i_1 \otimes i_2 \otimes i_2 \otimes i_1 \otimes acc \quad \wedge \quad a \sqsupseteq\sqsubseteq a \quad \bigwedge_j \quad r_j \sqsupseteq\sqsubseteq r_j \quad \wedge \\ & & i_1 \otimes i_1 \otimes i_2 \otimes i_2 \otimes i_1 \otimes acc \sqsubseteq i_1 \otimes acc \\ i_2 \sqsubseteq i_1 & \rightarrow & i_1 \otimes i_1 \otimes i_2 \otimes i_2 \otimes i_1 \otimes acc \sqsubseteq i_1 \otimes acc \quad \wedge \quad a \sqsubseteq a \quad \bigwedge_j \quad r_j \sqsubseteq r_j. \end{array}$$

Then the first precision between accumulators $i_1 \otimes acc \sqsubseteq i_1 \otimes i_1 \otimes i_2 \otimes i_2 \otimes i_1 \otimes acc$ follows from repeated applications of Equations (2) and (3), using that $i_1 \sqsubseteq i_2$, quasi-reflexivity of precision and self-precision of acc. The second precision between accumulators $i_1 \otimes i_1 \otimes i_2 \otimes i_2 \otimes i_1 \otimes acc \sqsubseteq i_1 \otimes acc$ follows the same pattern but uses Equations (1) and (3) instead. The third precision between accumulators $i_1 \otimes i_1 \otimes i_2 \otimes i_2 \otimes i_1 \otimes acc \sqsubseteq i_1 \otimes acc$ also follows the same pattern but uses Equations (2) and (3) and $i_2 \sqsubseteq i_1$, instead. The other precisions are given by self-precision hypothesis. □

Now we have everything we need to prove graduality for inductive families;

Theorem 6.6 (Graduality for Inductive Families). *Consider two indices $i_1, i_2 : \mathbf{Idx}^p$. If $i_1 \sqsubseteq i_2$, then $\langle \mathbb{I}\, p\, i_2 \Leftarrow \mathbb{I}\, p\, i_1 \rangle \dashv \langle \mathbb{I}\, p\, i_1 \Leftarrow \mathbb{I}\, p\, i_2 \rangle$ form an embedding projection pair.*

Proof. It follows from the definition of precision on constructors ($\mathbb{I}$-$\sqsubseteq$-$\mathbb{c}$), monotonicity of $\otimes$, and Lemma 6.5. □

Criteria for gradual indexed families. As we discussed in § 3.2, a correct treatment of gradual indexed inductive types must satisfy two properties: index relevance and shape relevance. These properties ensure that the index and shape of an inductive family are respected.

An indexed inductive type is index relevant when precision-related instances of an inductive imply relatedness of the indices.

Theorem 6.7 (Index relevance for inductive families). *Consider a parameter p, and two indices* $i_1, i_2 : \boldsymbol{Idx}^{p}$. *If* $\mathbb{I}\, p\, i_1 \sqsubseteq_\ell \mathbb{I}\, p\, i_2$, *then* $i_1 \;{}_{\boldsymbol{Idx}^{p_1}}\!\sqsubseteq_{\boldsymbol{Idx}^{p_2}}\; i_2$.

Proof. By the definition of precision for indexed inductive types. □

We establish that Punk's gradual indexed inductive types are shape relevant, i.e., they are only related by precision to indexed inductive types of the same family and to the exceptional types.

Theorem 6.8 (Shape relevance for inductive families). *Assuming weak-head normalization of types in Punk. Consider a parameter* $p : \boldsymbol{Param}$, *an index* $i : \boldsymbol{Idx}^{p}$, *and a type* $T : \square_\ell$.

$$\begin{array}{rl} & (T \sqsubseteq_\ell \mathbb{I}\, p\, i \rightarrow (T = \mathsf{err}_\square) \lor \exists (p' : \boldsymbol{Param})(i' : \boldsymbol{Idx}^{p'}).\, T = \mathbb{I}\, p'\, i') \\ \land & (\mathbb{I}\, p\, i \sqsubseteq_\ell T \rightarrow (T = ?_\square) \lor \exists (p' : \boldsymbol{Param})(i' : \boldsymbol{Idx}^{p'}).\, T = \mathbb{I}\, p'\, i') \end{array}$$

Proof. By the normalization assumption, we have that T reduces to $T' \in \mathbf{Whnf}_\square$. Then, the result is given by analysis on $\mathsf{head}\, T'$ and the rules for type precision of inductive families in Figure 8. □

Having discussed the general properties of Punk, we now turn our focus to properties specific to an index accumulator and indexed inductive types that uses them.

6.3 Characterizing Gradual Inductive Families

As we saw in §3, we can characterize gradual indexed inductive definitions by how casts commute with the `catch` eliminator up to precision. The asymmetry of the precision relation yield three cases of interest, each one potentially corresponding to a different class of inductive definitions. We show that each of the three classes arise from an index accumulator instance among those presented in §5.2.

The definitions and theorems in the rest of this section expect a monotone use of `catch`, therefore, we explicit all the assumptions here to avoid repeating them every time. Consider an inductive family $\mathbb{I} : \Pi(p : \mathbf{Param}), \mathbf{Idx}^{p} \rightarrow \square$; a parameter $p : \mathbf{Param}$; a motive $M : \Pi(i : \mathbf{Idx}^{p}), \mathbb{I}\, p\, i \rightarrow \square$; two indices $i_1, i_2 : \mathbf{Idx}^{p}$; branches $\overrightarrow{h_{\mathbb{c}_k}}$, h_{err} and h_{unk}; and term $t : \mathbb{I}\, p\, i_1$. All self-precise and if $h_{\mathsf{err}} \sqsubseteq h_{\mathbb{c}_k} \sqsubseteq h_{\mathsf{unk}}$ for all k. We do not fix the index accumulator since we will use a different one for each theorem. You can find the full theorems and more detailed proves in the appendix.

Definition 6.9 (Equiprecise elimination). An inductive family $\mathbb{I}$ satisfies equiprecise elimination if precision is preserved when casts are propagated outside of the eliminator:

$$\langle M\, p\, (\langle \mathbb{I}\, p\, i_2 \Leftarrow \mathbb{I}\, p\, i_1 \rangle\, t) \Leftarrow M\, p\, t \rangle\, \mathsf{catch}^{p}\, M\, \overrightarrow{h_{\mathbb{c}_k}}\, h_{\mathsf{err}}\, h_{\mathsf{unk}}\, i_1\, t \quad \sqsupseteq\!\sqsubseteq \quad \mathsf{catch}^{p}\, M\, \overrightarrow{h_{\mathbb{c}_k}}\, h_{\mathsf{err}}\, h_{\mathsf{unk}}\, i_2\, (\langle \mathbb{I}\, p\, i_2 \Leftarrow \mathbb{I}\, p\, i_1 \rangle\, t)$$

An example of an inductive family that satisfies equiprecise elimination is the free inductive family $\mathbb{I}_F$ using the initial index accumulator $\mathbf{Acc}_F$ (§5.1).

Theorem 6.10 (Free inductives satisfy equiprecise elimination).

$$\langle M\, p\, \langle \mathbb{I}_F\, p\, i_2 \Leftarrow \mathbb{I}_F\, p\, i_1 \rangle\, t \Leftarrow M\, p\, t \rangle\, \mathsf{catch}^{p}_{\mathbb{I}_F}\, M\, \overrightarrow{h_{\mathbb{c}_k}}\, h_{\mathsf{err}}\, h_{\mathsf{unk}}\, i_1\, t \sqsupseteq\!\sqsubseteq \mathsf{catch}^{p}_{\mathbb{I}_F}\, M\, \overrightarrow{h_{\mathbb{c}_k}}\, h_{\mathsf{err}}\, h_{\mathsf{unk}}\, i_2\, (\langle \mathbb{I}_F\, p\, i_2 \Leftarrow \mathbb{I}_F\, p\, i_1 \rangle\, t).$$

Proof. By induction on the typing derivation of t. The exceptional cases follow from the hypothesis and the reduction of cast on exceptions. The case of constructors is given by reduction of cast on constructors and the fact that $\mathbb{r}$ is the identity for `list`. □

Definition 6.11 (Over-precise elimination). An inductive family $\mathbb{I}$ satisfies overprecise elimination if precision increases when casts are propagated outside of the eliminator.

$$\langle M\,p\,(\langle \mathbb{I}\,p\,i_2 \Leftarrow \mathbb{I}\,p\,i_1\rangle\,t) \Leftarrow M\,p\,t\rangle\,\mathtt{catch}^p\,M\,\overrightarrow{h_{\mathtt{c}_k}}\,h_{\mathsf{err}}\,h_{\mathsf{unk}}\,i_1\,t \quad \boxed{\sqsubseteq} \quad \mathtt{catch}^p\,M\,\overrightarrow{h_{\mathtt{c}_k}}\,h_{\mathsf{err}}\,h_{\mathsf{unk}}\,i_2\,(\langle \mathbb{I}\,p\,i_2 \Leftarrow \mathbb{I}\,p\,i_1\rangle\,t)$$

An example of an inductive family that satisfies underprecise elimination is the forgetful inductive family $\mathbb{I}_U$ using the index accumulator $\mathbf{Acc}_U^p$ from §5.2.

Theorem 6.12 (Forgetful inductives satisfy overprecise elimination).

$$\langle M\,p\,\langle \mathbb{I}_U\,p\,i_2 \Leftarrow \mathbb{I}_U\,p\,i_1\rangle\,t \Leftarrow M\,p\,t\rangle\,\mathtt{catch}^p_{\mathbb{I}_U}\,M\,\overrightarrow{h_{\mathtt{c}_k}}\,h_{\mathsf{err}}\,h_{\mathsf{unk}}\,i_1\,t \sqsubseteq \mathtt{catch}^p_{\mathbb{I}_U}\,M\,\overrightarrow{h_{\mathtt{c}_k}}\,h_{\mathsf{err}}\,h_{\mathsf{unk}}\,i_2\,(\langle \mathbb{I}_U\,p\,i_2 \Leftarrow \mathbb{I}_U\,p\,i_1\rangle\,t).$$

Proof. By induction on the typing derivation of t. The exceptional cases follow from the hypothesis and the reduction of cast on exceptions. The case of constructors goes by induction on the index accumulator *acc*. If (*acc* = none), then the result follows from the self-precision hypotheses. On the other hand, if (*acc* = some i'), by reduction of `catch` on constructors, we have to show

$$\begin{aligned} &\langle M\,i_2\,(\mathtt{c}^p(\mathsf{some}\ i_2)\,a\,\overrightarrow{r_j}) \Leftarrow M\,i_1\,(\mathtt{c}^p(\mathsf{some}\ i_1)\,a\,\overrightarrow{r_j}) \Leftarrow M\,(\mathbf{ctr\_idx}^p_{\mathtt{c}}\,a)\,(\mathtt{c}^p\mathsf{none}\,a\,\overrightarrow{r_j})\rangle\,t' \\ \sqsubseteq\ &\langle M\,i_2\,(\mathtt{c}^p(\mathsf{some}\ i_2)\,a\,\overrightarrow{r_j}) \Leftarrow M\,(\mathbf{ctr\_idx}^p_{\mathtt{c}}\,a)\,(\mathtt{c}^p\mathsf{none}\,a\,\overrightarrow{r_j})\rangle\,t' \end{aligned}$$

where $t' \triangleq \mathtt{catch}^p_{\mathbb{I}_U}\,M\,\overrightarrow{h_{\mathtt{c}_k}}\,h_{\mathsf{err}}\,h_{\mathsf{unk}}\,(\mathbf{ctr\_idx}^p_{\mathtt{c}}\,a)\,(\mathtt{c}^p\mathsf{none}\,a\,\overrightarrow{r_j})$. Which is given by cast decomposition, counit, and self-precision of hypotheses. □

Definition 6.13 (Under-precise elimination). An inductive family $\mathbb{I}$ satisfies underprecise elimination if precision decreases when casts are propagated outside of the eliminator.

$$\langle M\,p\,(\langle \mathbb{I}\,p\,i_2 \Leftarrow \mathbb{I}\,p\,i_1\rangle\,t) \Leftarrow M\,p\,t\rangle\,\mathtt{catch}^p\,M\,\overrightarrow{h_{\mathtt{c}_k}}\,h_{\mathsf{err}}\,h_{\mathsf{unk}}\,i_1\,t \quad \boxed{\sqsupseteq} \quad \mathtt{catch}^p\,M\,\overrightarrow{h_{\mathtt{c}_k}}\,h_{\mathsf{err}}\,h_{\mathsf{unk}}\,i_2\,(\langle \mathbb{I}\,p\,i_2 \Leftarrow \mathbb{I}\,p\,i_1\rangle\,t)$$

An example of an inductive family that satisfies underprecise elimination is the forgetful inductive family $\mathbb{I}_M$ using the index accumulator $\mathbf{Acc}_M^p$ from §5.2.

Theorem 6.14 (Meet inductives satisfy underprecise elimination).

$$\mathtt{catch}^p_{\mathbb{I}_M}\,M\,\overrightarrow{h_{\mathtt{c}_k}}\,h_{\mathsf{err}}\,h_{\mathsf{unk}}\,i_2\,(\langle \mathbb{I}_M\,p\,i_2 \Leftarrow \mathbb{I}_M\,p\,i_1\rangle\,t).\ \sqsubseteq \langle M\,p\,\langle \mathbb{I}_M\,p\,i_2 \Leftarrow \mathbb{I}_M\,p\,i_1\rangle\,t \Leftarrow M\,p\,t\rangle\,\mathtt{catch}^p_{\mathbb{I}_M}\,M\,\overrightarrow{h_{\mathtt{c}_k}}\,h_{\mathsf{err}}\,h_{\mathsf{unk}}\,i_1\,t$$

Proof. By induction on the typing derivation of $t^{\sqsubseteq_{\mathbb{I}\,p\,i}}$. The exceptional cases follow from the hypothesis and the reduction of cast on exceptions. The case for constructors, follows by induction on the index accumulator *acc*. For (*acc* = zero), the result is given by retraction and self-precision of M. If (*acc* = two i_1 i'), it is given by precision rule $\mathbf{Acc}_M^p$ ($\mathbf{Acc}_M$-$\sqsubseteq$-two) and properties of the meet index accumulator. □

7 Related Work

Having given a formal presentation of Punk we can now revisit in more detail the comparison with previous approaches discussed in §2.3. Then, we briefly discuss the related work regarding effectful and gradual dependent types (§7.2 and §7.3).

7.1 Previous Approaches to Gradual Indexed Inductive Types

GCIC. Lennon-Bertrand et al. [2022] present an implementation of vectors for GCIC where casts are eagerly reduced and casting constructors to the unknown index is internalized as a new constructor. In particular, casting a constructor to an incompatible index immediately fails in GCIC:

$$\langle \mathbb{V}\,B\,(\mathsf{S}\,n) \Leftarrow \mathbb{V}\,A\,0\rangle\,\mathtt{nil} \rightsquigarrow_{\mathtt{GCIC}} \mathtt{err}_{\mathbb{V}\,B\,(\mathsf{S}\,n)}$$

This behavior cannot be encoded in PUNK, because cast reduction always preserves the head constructor, regardless of the choice of index accumulator. For example, the cast reduction rule for constructors (GI-CTOR-CAST) can be instantiated to the nil constructor for $\mathbb{V}$ as

$$\langle \mathbb{V}\, B\, n \Leftarrow \mathbb{V}\, A\, m\rangle\, \mathtt{nil}\; acc \rightsquigarrow \mathtt{nil}\; (\mathsf{n} \otimes (\langle \mathbf{Acc}^{\mathsf{B}} \Leftarrow \mathbf{Acc}^{\mathsf{A}}\rangle\, \mathsf{m} \otimes acc)).$$

Note that the reduction only updates the index accumulator, so casting nil always reduces to nil, and cannot fail; errors only appear on elimination with catch. This difference is justified by the fact that matching on the index of the target of a cast is not possible in general (e.g., when the index is a function or a type). Indeed, the authors informally argue that this approach generalizes to other inductive families as long as their indices are forceable [Brady et al. 2004]. This in particular leaves out propositional equality, discussed next. As a general framework for the definition of indexed inductive types, PUNK's cast reduction semantics must accommodate indices of any type.

GEq. Eremondi et al. [2022] develop GEq, an extension of GCIC with a type former for propositional equality ($t_1 =^{\mathsf{GEq}}_{T} t_2$) inhabited by $\mathsf{refl}^{\mathsf{GEq}}_{t_w \vdash t_1 \cong t_2}$, which holds a witness terms t_w more precise than both t_1 and t_2. The typing rule for refl (CASTREFL) is defined using bidirectional typing, following Lennon-Bertrand et al. [2022], and the precision judgment $\Gamma | \Gamma' \vdash t \sqsubseteq^{\leftarrow}_{\rightarrow} t'$, which states that t is more precise than t' up-to conversion, with t and t' well-typed in contexts Γ and Γ' respectively:

$$\textsc{CastRefl}\quad \frac{\Gamma \vdash t_w \Rightarrow T \qquad \Gamma \vdash t_1 \Leftarrow T \qquad \Gamma \vdash t_2 \Leftarrow T \qquad \Gamma|\Gamma \vdash t_w \sqsubseteq^{\leftarrow}_{\rightarrow} t_1 \qquad \Gamma|\Gamma \vdash t_w \sqsubseteq^{\leftarrow}_{\rightarrow} t_2}{\Gamma \vdash \mathsf{refl}^{\mathsf{GEq}}_{t_w \vdash t_1 \cong t_2} \Rightarrow t_1 =^{\mathsf{GEq}}_{T} t_2}$$

Instead of a general catch, propositional equality in GEq is equipped with a custom eliminator $\mathsf{J}(M, t_1, t_2, t_{M1}, t)$ that, given an equality t with type $t_1 =^{\mathsf{GEq}}_{T} t_2$, transforms a term t_{M1} of type $M\, t_1$ into one of type $M\, t_2$. Similarly to meet indexed inductive types in PUNK, casts in GEq accumulate the meet of the source and target indices (REDCASTEQ) and elimination propagates the casts to the outside (REDJ):

$$\textsc{RedCastEq}\colon \langle t_1' =^{\mathsf{GEq}}_{T'} t_2' \Leftarrow t_1 =^{\mathsf{GEq}}_{T} t_2\rangle\, \mathsf{refl}^{\mathsf{GEq}}_{t_w \vdash t_1 \cong t_2} \rightsquigarrow_{\mathsf{GEq}} \mathsf{refl}^{\mathsf{GEq}}_{(\langle T' \Leftarrow T\rangle\, t_w \sqcap t_1' \sqcap t_2') \vdash t_1' \cong t_2'}$$

$$\textsc{RedJ}\colon \mathsf{J}(M, t_1, t_2, t_{M1}, \mathsf{refl}^{\mathsf{GEq}}_{t_w \vdash t_1 \cong t_2}) \rightsquigarrow_{\mathsf{GEq}} \langle M\, t_2 \Leftarrow M\, t_w\rangle\, \langle M\, t_w \Leftarrow M\, t_1\rangle\, t_{M1}$$

We can encode the J eliminator in PUNK as $\mathsf{J}_{\textsc{Punk}}$ which satisfies the REDJ reduction rule:

$$\lambda T. \mathsf{J}_{\textsc{Punk}}(M, t_1, t_2, t_{M1}, t) \triangleq \mathtt{catch}^{T, t_1}\, M\, (\lambda t_2'. \langle M\, t_2' \Leftarrow M\, t_1\rangle\, t_{M1})\, (\lambda t_2'. \langle M\, t_2' \Leftarrow M\, t_1\rangle\, t_{M1})\, t_{M1}\, t_2\, t$$

But, in contrast to PUNK, GEq conflates exceptions with refl, the only canonical inhabitant of equality (PROPEQUNK and PROPEQERR):

$$\textsc{PropEqUnk}\colon ?_{t_1 =^{\mathsf{GEq}}_{T} t_2} \rightsquigarrow_{\mathsf{GEq}} \mathsf{refl}^{\mathsf{GEq}}_{t_1 \sqcap t_2 \vdash t_1 \cong t_2} \qquad \textsc{PropEqErr}\colon \mathtt{err}_{t_1 =^{\mathsf{GEq}}_{T} t_2} \rightsquigarrow_{\mathsf{GEq}} \mathsf{refl}^{\mathsf{GEq}}_{\mathtt{err}_T \vdash t_1 \cong t_2}$$

We observe that this construction is specific to propositional equality and cannot be generalized to inductive families with multiple constructors: in order to preserve err and ? as the bottom and top elements of the precision relation, supporting this behavior in general requires every pair of constructors to be related with each other.

Furthermore, because there is no notion of empty index accumulator, the embedding of CIC into GEq translates refl t into $\mathsf{refl}^{\mathsf{GEq}}_{t \vdash t \cong t}$ which introduces extra identity casts. This means that a conservativity result over CIC (as in Theorem 6.4) would require an extra conversion rule for identity casts ($\langle A \Leftarrow A\rangle\, a \equiv a$); it is unclear how to achieve this in a gradual setting, although Pujet and Tabareau [2024] achieve this for observational equality.

Note that, as mentioned in § 2.3, it is possible to use propositional equality to define indexed inductive types via fording. In fact, one can obtain a cast reduction semantics similar to that induced by a given index accumulator by using fording with a propositional equality with that index accumulator. A deeper and formal exploration of this connection is future work.

7.2 Effects in Dependent Type Theory

Gradual type systems rely on effects, in particular errors, to enforce dynamic typing invariants, a challenging aspect within type theory. Indeed, the Fire Triangle of Pédrot and Tabareau [2020] exhibit a strong tension between the metatheoretic properties of CIC, starting with logical consistency, and observable effects. Dependently typed programming languages featuring effectful computations either isolate dependencies to a fragment of pure expressions [Swamy et al. 2016; Xi and Pfenning 1998] or give up on some metatheoretical properties: Dependent Haskell [Eisenberg 2016], Trellys [Kimmell et al. 2012; Stump et al. 2010] and Zombie [Casinghino et al. 2014] which allow diverging type-level expressions either admit inconsistencies or undecidable typechecking. The recent work of Pédrot and Tabareau [2017, 2018] specifically consider exceptions in type theory, building up to RETT, a type theory exploiting universe hierarchies to separate effectful, inconsistent computations from pure, consistent proofs to reason about the effectful content [Pédrot et al. 2019].

7.3 Gradual Dependent Types

This work pursue 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. 2015a], 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] 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, and propose approximate normalization to ensure decidable typechecking. Gradual inductive types only appear in GCIC [Lennon-Bertrand et al. 2022], and its follow up, GRIP [Maillard et al. 2022], upon which we build Punk.

8 Conclusion

Punk provides a general framework to define gradual indexed inductive families. As illustrated in §6.3, Punk is able to explore the space of cast reduction semantics, providing adequate descriptions for different semantics of indexed inductive types in a unified way. Its expressivity stems from the use of a simple and algebraic notion of listable indexed accumulators taking, in a gradual setting, the role that would otherwise be filled by propositional equality in approaches of indexed inductive types using Fordism in CIC. This abstraction not only allows to prove the metatheory of the framework uniformly for all possible semantics of gradual indexed inductive types, but also open the way for programmers using gradual dependent types to choose the implementation of gradual indexed inductive types that best fit their need, arbitrating between space efficiency and precision of the dynamic typing constraint tracking.

References

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. https://doi.org/10.1145/2837614

Edwin Brady. 2013. Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation. *Journal of Functional Programming* 23, 5 (Sept. 2013), 552–593.

Edwin Brady, Conor McBride, and James McKinna. 2004. *Types for Proofs and Programs.* Lecture Notes in Computer Science, Vol. 3085. Springer-Verlag, Chapter Inductive Families Need Not Store Their Indices, 115–129.

Ali Caglayan. 2021. https://github.com/HoTT/Coq-HoTT/blob/master/theories/Types/IWType.v

Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. 2014. Combining proofs and programs in a dependently typed language. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014).* ACM Press, San Diego, CA, USA, 671–684. https://doi.org/10.1145/2535838.2535883

Giuseppe Castagna (Ed.). 2009. *Proceedings of the 18th European Symposium on Programming Languages and Systems (ESOP 2009).* Lecture Notes in Computer Science, Vol. 5502. Springer-Verlag, York, UK.

Adam Chlipala. 2013. *Certified Programming with Dependent Types.* MIT Press.

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

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction (Lecture Notes in Computer Science)*, Amy P. Felty and Aart Middeldorp (Eds.). Springer International Publishing, Cham, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

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

Joseph Eremondi, Ronald Garcia, and Éric Tanter. 2022. Propositional Equality for Gradual Dependently-Typed Programming. See[ICFP 2022 2022], 165–193.

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

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing, See [Bodík and Majumdar 2016], 429–442. https://doi.org/10.1145/2837614 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

Michael Greenberg. 2017. Space-Efficient Manifest Contracts. arXiv:1410.2813 [cs.PL] https://arxiv.org/abs/1410.2813

ICFP 2019. *Proceedings of the 24th ACM SIGPLAN Conference on Functional Programming (ICFP 2019).* Vol. 3. ACM Press.

ICFP 2022 2022.

Garrin Kimmell, Aaron Stump, Harley D. Eades III, Peng Fu, Tim Sheard, Stephanie Weirich, Chris Casinghino, Vilhelm Sjöberg, Nathan Collins, and Ki Yung Ahn. 2012. Equational reasoning about programs with general recursion and call-by-value semantics. In *Proceedings of the 6th workshop on Programming Languages Meets Program Verification (PLPV 2012).* ACM Press, 15–26. https://doi.org/10.1145/2103776.2103780

Kenneth Knowles and Cormac Flanagan. 2010. Hybrid type checking. *ACM Transactions on Programming Languages and Systems* 32, 2 (Jan. 2010), Article n.6. https://doi.org/10.1145/1111037.1111059

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), 7:1–7:82.

Kenji Maillard, Meven Lennon-Bertrand, Nicolas Tabareau, and Éric Tanter. 2022. A Reasonably Gradual Type Theory. See[ICFP 2022 2022], 931–959.

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

Conor McBride. 1999. *Dependently Typed Functional Programs and their Proofs.* Ph. D. Dissertation. University of Edinburgh.

Max S. New and Amal Ahmed. 2018. Graduality from Embedding-Projection Pairs, In Proceedings of the 23rd ACM SIGPLAN Conference on Functional Programming (ICFP 2018). *Proceedings of the ACM on Programming Languages* 2, 73:1–73:30. https://doi.org/10.1145/3236768

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.

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. https://doi.org/10.1145/2103776.2103779

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. https://doi.org/10.1007/1-4020-8141-3_34

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. https://doi.org/10.1007/978-3-319-89884-1_9

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. https://doi.org/10.1145/3371126

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

Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjoberg, and Brent Yorgey. 2015. *Software Foundations*. Electronic textbook. http://www.cis.upenn.edu/ bcpierce/sf.

POPL 2010 2010. *Proceedings of the 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010)*. ACM Press, Madrid, Spain.

Loïc Pujet and Nicolas Tabareau. 2024. Observational Equality Meets CIC. In *Programming Languages and Systems: 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6–11, 2024, Proceedings, Part I* (Luxembourg City, Luxembourg). Springer-Verlag, Berlin, Heidelberg, 275–301. https://doi.org/10.1007/978-3-031-57262-3_12

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. https://doi.org/10.1145/1375581.1375602

Jeremy Siek, Ronald Garcia, and Walid Taha. 2009. Exploring the Design Space of Higher-Order Casts, See [Castagna 2009], 17–31.

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, See [POPL 2010 2010], 365–376. https://doi.org/10.1145/1706299.1706342

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015a. 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. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015b. Monotonic References for Efficient Gradual Typing. In *Proceedings of the 24th European Symposium on Programming Languages and Systems (ESOP 2015) (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer-Verlag, London, UK, 432–456.

Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 2019. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. 4, POPL, Article 8 (dec 2019), 28 pages. https://doi.org/10.1145/3371076

Aaron Stump, Vilhelm Sjöberg, and Stephanie Weirich. 2010. Termination Casts: A Flexible Approach to Termination with General Recursion. In *Proceedings Workshop on Partiality and Recursion in Interactive Theorem Provers (PAR 2010)*. 76–93. https://doi.org/10.29007/3w36

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. https://doi.org/10.1145/2837614

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.

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed, See [Castagna 2009], 1–16. https://doi.org/10.1007/978-3-642-00590-9_1

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. https://doi.org/10.1145/277650.277732

Received 2024-02-28; accepted 2024-06-18