1

A Reasonably Exceptional Type Theory

PIERRE-MARIE PÉDROT, Inria, France
NICOLAS TABAREAU, Inria, France
HANS JACOB FEHRMANN, University of Chile, Chile
ÉRIC TANTER, University of Chile, Chile and Inria Paris, France

Traditional approaches to compensate for the lack of exceptions in type theories for proof assistants have severe drawbacks from both a programming and a reasoning perspective. Pédrot and Tabareau recently extended the Calculus of Inductive Constructions (CIC) with exceptions. The new exceptional type theory is interpreted by a translation into CIC, covering full dependent elimination, decidable type-checking and canonicity. However, the exceptional theory is inconsistent as a logical system. To recover consistency, Pédrot and Tabareau propose an additional translation that uses parametricity to enforce that all exceptions are caught locally. While this enforcement brings logical expressivity gains over CIC, it completely prevents reasoning about exceptional programs such as partial functions. This work addresses the dilemma between exceptions and consistency in a more flexible manner, with the *Reasonably Exceptional Type Theory* (RETT). RETT is structured in three layers: (a) the *exceptional* layer, in which all terms can raise exceptions; (b) the *mediation* layer, in which exceptional terms must be provably parametric; (c) the *pure* layer, in which terms are non-exceptional, but can refer to exceptional terms. We present the general theory of RETT, where each layer is realized by a predicative hierarchy of universes, and develop an instance of RETT in Coq: the impure layer corresponds to the predicative universe hierarchy, the pure layer is realized by the impredicative universe of propositions, and the mediation layer is reified via a parametricity type class. RETT is the first full dependent type theory to support consistent reasoning about exceptional terms, and the CoqRETT plugin readily brings this ability to Coq programmers.

1 FAILURE IN TYPE THEORY

The absolute purity of type theories like the Calculus of Constructions [Coquand and Huet 1988] is both a blessing and a curse. A blessing because purity implies consistency of the internal logic, thereby validating their use as foundations of proof assistants such as Coq [The Coq Development Team 2019] and Agda [Norell 2009], within which one can express and prove interesting mathematical results, including about programs and programming languages. A curse because the lack of a basic effect like failure makes the use of these theories in practical scenarios, in particular in their dual use as functional programming languages, cumbersome at best.

The Failure Problem. As a matter of fact, many common situations would benefit from a convenient way to deal with partiality or failure, like exceptions in mainstream programming languages. We will call this the *failure problem*. Traditional solutions to the failure problem are monadic programming, default values, and axioms; each of which has severe drawbacks as discussed next.

Monadic Programming. The standard approach to the failure problem in functional programming is to use the option (or exception) monad, in which values are tagged explicitly to indicate whether they denote a success or a failure. For instance, the head function on lists can be given type $\Pi A : \square.\, \texttt{list}\ A \to \texttt{option}\ A$, where $\square$ is the universe (a.k.a. kind) of types. This approach is notoriously contagious, widely imposing a monadic style of programming. More problematic, while it can be used without too much pain in a non-dependently-typed setting like e.g. in Haskell, the monadic approach does not scale well to dependent types.

Authors' addresses: Pierre-Marie Pédrot, Inria, Gallinette Project-Team, Nantes, France; Nicolas Tabareau, Inria, Gallinette Project-Team, Nantes, France; Hans Jacob Fehrmann, University of Chile, Computer Science Department (DCC), Santiago, Chile; Éric Tanter, University of Chile, Computer Science Department (DCC), Santiago, Chile , Inria Paris, France.

For instance, if a function f returns an optional value, then even a simple dependent property like $\forall x. x > 0 \to f\ x > 0$ needs to be stated with type-level pattern matching, such as: $\forall x. x > 0 \to$ `match` $f\ x$ `with Some` $y \to y > 0 \,|\,$ `None` $\to$???. In addition to quickly becoming untractable, this technique is also not systematic: what type should one put in place of ??? in the failure branch? For positive occurences, a top type seems useful, but dually, for negative occurences, a bottom type should be considered to rule out exceptional cases.

Default Values. Due to these issues, many libraries tend to favor the use of default values over the exception monad. Default values preserve simple signatures, but for polymorphic functions, coming up with a default value is tricky, requiring either the use of type class resolution to automatically infer default values for certain (inhabited) types—an approach used for instance in hs-to-coq [Breitner et al. 2018]—or changing the signature of functions to require as first argument a default value to return in case of failure—an approach favored in the Mathematical Components library [Mahboubi and Tassi 2008], where e.g. head : $\Pi A : \square.\ A \to \texttt{list}\ A \to A$.

A major drawback of using default values is the potential confusion with normal values: if head returns the default value, is it because the list was empty, or because the default value *was* actually the head of the list? Consequently, reasoning about such artificially total functions is compromised. Likewise, how to state that "the tail of a non-empty list does not fall through the problematic branch"? The property

$$\texttt{length}\ l > 0 \to \texttt{tail}\ d\ l \neq d$$

is not true in general: it does not hold if d is the empty list and l has just one element.

Axiomatic Approach. To avoid imposing a monadic style while avoiding confusion of values, another approach is to use axioms to denote exceptions, as explored by Tanter and Tabareau [2015] in their cast framework for subset types. The problem of axioms is that they have no computational content, therefore raising an exception materializes as a stuck term; it is impossible to catch such axiomatic exceptions and to reason about potential failing terms. For instance, if `tail` uses an axiom `error` when applied to an empty list, the property

$$\texttt{length}\ l > 0 \to \texttt{tail}\ l \neq \texttt{error}$$

is not provable, because one is not allowed discriminate the axiom from pure terms.

Exceptional Type Theory. Recently, Pédrot and Tabareau [2018] developed an extension of CIC with exceptions, interpreted by a translation into CIC, and implemented in Coq as a plugin. The Exceptional Type Theory (ETT) includes a function `fail` : $\Pi A : \square.\ A$ that throws an exception at any type (for readability, we omit the type argument in the remainder of this section). This function enjoys the computational behavior that one would expect, namely that the exception escapes from contexts that evaluate it.

This solves the failure problem in a straightforward way. ETT makes it possible to define head and `tail` functions that raise exceptions when applied to the empty list, without polluting their type signatures, and to prove that

$$\vdash_{\mathrm{ETT}} \texttt{length}\ l > 0 \to \texttt{tail}\ l \neq \texttt{fail}.$$

The good news is that ETT is computationally relevant, that is, programs reduce to normal forms and the equational theory is not degenerate. As such, it can be used as a dependently-typed programming language with exceptions.

The flip side is that ETT is inconsistent as a logic. Just as in any programming language featuring exceptions, it is indeed possible to inhabit any type, and thus any property, by raising an exception. In particular, ETT *also* allows one to prove the paradoxical fact that

$$\vdash_{\mathrm{ETT}} \texttt{length}\ l > 0 \to \texttt{tail}\ l = \texttt{fail}.$$

Peeking at the proofs of those two properties reveals that they are not anywhere near being equally valid, though. The first one is correct in the sense that it is only using the exception-free fragment of ETT, while the second one is a blatant lie, as executing it would lead to an immediate dynamic failure.

By defining a subset of *valid* proofs, it is possible to recover logical consistency. Pédrot and Tabareau [2018] give a second interpretation of CIC as a restriction of ETT, using parametricity [Bernardy and Lasson 2011] to force all exceptions to be locally handled. This approach is useful to extend the logical expressivity of CIC with a kind of backtracking-based reasoning. Unfortunately, the restriction is too strong and is not applicable to the programming setting considered here. One cannot define exception-raising functions like `head` or `tail` anymore, because by construction they do not satisfy the validity criterion.

Consistent Reasoning about Exceptional Programs. The contribution of this paper is to present a new type theory, dubbed the Reasonably Exceptional Type Theory (RETT), which supports consistent reasoning about exceptional programs. The core feature of RETT that makes this possible is a universe-based separation between consistent proofs and effectful programs. This split is embodied by the existence of parallel hierarchies of safe vs. unsafe types that are allowed to interact in a principled way.

We implement CoqRETT, a fragment of RETT in Coq as a plugin. Seizing their similarity of purpose, CoqRETT piggybacks on Coq's `Prop`-`Type` classification to separate consistent logical reasoning (in `Prop`) from effectful programmming (in `Type`). We convey a foretaste of CoqRETT in the paragraphs below.

In CoqRETT, we can define `head` and `tail` as functions that raise exceptions when applied to empty lists, as those terms live in the `Type` hierarchy. We can then prove as expected

$$\texttt{length}\ l > 0 \rightarrow \texttt{tail}\ l \neq \texttt{fail}.$$

Contrarily to ETT, in CoqRETT the paradoxical proposition $\texttt{length}\ l > 0 \rightarrow \texttt{tail}\ l = \texttt{fail}$ does not hold, because equality lives in `Prop`, forbidding inconsistent reasoning. Similarly and somehow counter-intuitively, in CoqRETT we cannot prove that

$$\Pi n : \mathbb{N}.\, n \geq 0.$$

assuming $\geq$ is defined in the usual way. The reason is that exceptions also inhabit the `Type`-dwelling type of natural numbers $\mathbb{N}$, while $\geq$ only mentions pure integers. In order to be able to reason about terms that are actually pure, we need to introduce a parametricity predicate `param`, realized using a Coq type class. This way, we can prove

$$\Pi n : \mathbb{N}.\, \texttt{param}\ n \rightarrow n \geq 0.$$

This selective and explicit approach to parametricity is the key to allow both consistent reasoning and exceptional terms to coexist.

Pédrot and Tabareau [2018] observe that exceptions in type theory are naturally call-by-name exceptions. This means for instance that there are multiple levels at which "being a list of natural numbers" can interact with failure: the whole list can be an exception, the spine of the list can be a proper structure but it can have exceptions as elements (i.e. fake natural numbers), or the list can be a pure, deeply parametric list.

To illustrate, consider a property of an exception-raising `head` function, namely that it does not fail when applied to non-empty lists

$$\texttt{length}\ l > 0 \rightarrow \texttt{head}\ l \neq \texttt{fail}.$$

Stated in this way, this property is in fact false. Is it because l itself could be an exception? No. In fact, we can prove that $\texttt{length}\ l > 0 \rightarrow l \neq \texttt{fail}$, because if $l = \texttt{fail}$, then $\texttt{length}\, l$ is convertible to `fail`; but the proposition `fail > 0` cannot be proven in `Prop`, which is consistent. In other words, `length` $l > 0 \rightarrow$ `param` l.

But even though l is a "real" list, its elements might not be. In particular, the first element might be an exception, thereby negating the property above. In order to properly state the property, we therefore need a deep version of the parametricity predicate. We can then prove that

$$\mathtt{param_{deep}}\ l \rightarrow \mathtt{length}\ l > 0 \rightarrow \mathtt{head}\ l \neq \mathtt{fail}.$$

In brief, CoqRETT allows programmers to use exceptions in their programs, and to consistently reason about them at the required level of granularity, accounting for potential failures extrinsically and when needed. We come back to these examples in Section 5.3, presenting in detail their statements and proofs in CoqRETT.

A Reasonably Exceptional Type Theory. CoqRETT represents the practical contribution of this work. However, as the illustration above reveals, CoqRETT relies in an essential way on the parametricity predicate `param`, which is realized through a type class. This is problematic from a foundational point of view, because type classes and ad hoc polymorphism cannot be directly accounted for in a type-theoretic setting.

To put consistent reasoning about exceptional terms on a solid type theoretic footing, we propose the Reasonably Exceptional Type Theory (RETT). RETT features three separate universe hierarchies, which can be thought of as adjacent layers: one exceptional, one pure, and in between a mediation layer in which parametricity is realized (Section 3). Modalities, defined as functions, coordinate the interplay between these layers. We give a syntactic model of RETT by translation into CIC, interpreting each layer and modality in a specific way (Section 4). This translation allows us to prove the metatheoretical properties of RETT. CoqRETT is then formally justified by considering a fragment of RETT that is implementable in Coq (Section 5) without having to modify its kernel. Coq being restricted to two universe hierarchies, we map the exceptional layer to the predicative hierarchy (`Type`), and the pure layer to the impredicative universe (`Prop`); type classes are then an implementation technique to reify the parametricity predicate from the (missing) mediation layer.

The implementation of the CoqRETT plugin and the examples discussed here are provided in supplementary material; they have been tested in Coq 8.8.

2 BACKGROUND: EXCEPTIONAL TRANSLATION AND PARAMETRICITY

We first provide a quick introduction to the key technical ideas of the Exceptional Type Theory (ETT) of Pédrot and Tabareau [2018], on which our technical development is based. We recall both interpretations of ETT: the standard exceptional translation, which yields a logically inconsistent theory; and the parametric exceptional translation, which recovers consistency through parametricity, at the expense of expressiveness.

2.1 Exceptional Translation

As mentioned in the introduction, ETT is an an extension of CIC with exceptions. ETT includes an exception type $\mathbf{E} : \square$ and a function $\mathtt{raise} : \Pi A : \square.\ \mathbf{E} \rightarrow A$ to raise exceptions at any type A. ETT is justified by a syntactic translation into CIC, denoted $[M]$ for any ETT term M, which is a simplification of the weaning translation of Pédrot and Tabareau [2017]. Intuitively, a type A in ETT is interpreted as a pair of a type A in CIC together with a default function $A_\varnothing : \mathbb{E} \rightarrow A$ specifying how to interpret failure on this type. Here, $\mathbb{E}$ is the CIC representation type of the source exception type $\mathbf{E}$.

Because the universe of types is itself a type, one needs to define a representation for types that can raise exceptions. This can be done with the following inductive type:

$$\begin{aligned}&\texttt{Ind type}_i \; : \square_{i+1} := \\ &\mid \texttt{TypeVal}_i : \Pi A : \square_i.\, (\mathbb{E} \to A) \to \texttt{type}_i \\ &\mid \texttt{TypeErr}_i : \mathbb{E} \to \texttt{type}_i\end{aligned}$$

The constructor $\texttt{TypeVal}_i$ constructs a $\texttt{type}_i$ from a type and a default function on this type. The constructor $\texttt{TypeErr}_i$ represents the default function at the level of $\texttt{type}_i$. The exceptional translation uses a term $\texttt{El}_i : \texttt{type}_i \to \square_i$ to recover the underlying type from an inhabitant of $\texttt{type}_i$, and a term $\texttt{Err}_i : \Pi A : \texttt{type}_i.\, \mathbb{E} \to \texttt{El}_i\, A$ to lift the default function to this underlying type.

The translation of an ETT universe is therefore a value of the above inductive type:

$$[\square_i] := \texttt{TypeVal}_{i+1}\; \texttt{type}_i\; \texttt{TypeErr}_i$$

The ETT exception type $\mathbf{E}$ is mapped to $\mathbb{E}$ together with the identity as default function:

$$[\mathbf{E}] := \texttt{TypeVal}\; \mathbb{E}\; (\lambda e : \mathbb{E}.\, e)$$

and the function `raise` raises the provided exception at any type as:

$$[\texttt{raise}] := \lambda (A : \texttt{type})\, (e : \mathbb{E}).\, \texttt{Err}\; A\; e$$

For inductive types, the translation freely adds an additional constructor, similarly to `TypeErr` for the universe. For instance, the translation of the inductive type of booleans $\mathbb{B}$ is

$$[\mathbb{B}] := \texttt{TypeVal}\; \mathbb{B}^\bullet\; \mathbb{B}_\varnothing$$

where $\mathbb{B}^\bullet$ is an inductive type with three constructors: $\texttt{true}^\bullet$, $\texttt{false}^\bullet$ and an extra default constructor $\mathbb{B}_\varnothing : \mathbb{E} \to \mathbb{B}^\bullet$.

Observe that this treatment of inductive types means that the empty type of ETT is translated to an inductive with one (default) constructor. Therefore the empty type is inhabited as soon as the target exception type $\mathbb{E}$ is. In fact, any proof of the empty type is an exception, which means that one can use exceptions to prove any result. ETT is inconsistent as a logic.

2.2 Exceptional Parametric Translation

To recover logical consistency, Pédrot and Tabareau [2018] give a second interpretation of ETT that uses the standard parametricity translation for type theory [Bernardy and Lasson 2011] in addition to the exceptional translation.

Let us first recall the essence of the (unary) parametricity translation for type theory. While in System F, parametricity has to be stated and proved externally, the expressiveness of type theory makes it possible to internalize the parametricity argument as a translation from terms to terms. The mere fact that the translation is defined for all terms means that these terms are parametric—this property is known as the *abstraction theorem* [Reynolds 1983]. In type theory, for the universe, the parametricity translation is defined as arbitrary predicates on types, e.g. type A is translated to a predicate of type $A \to \square$. These predicates are called *parametricity predicates*, inhabited by *parametricity witnesses*. For the dependent function type $\Pi x : A.\, B$, the translation specifies that a *valid* input, i.e. an argument of type A together with its parametricity witness, yields a *valid* output at type B. Consequently, the translation of a lambda term is a function takes an argument and a parametricity witness, and an application is translated so as to pass the parametricity witness as extra argument.

In the context of ETT, one can simply use the parametricity translation to detect pure terms by postulating the *absence* of any parametricity witness for `raise`. In this way, given a term, if its parametricity translation is defined, then the term does not use `raise`. While this violently rules out any use of exceptions, one can manually extend the parametricity translation for a term t that uses exceptions *internally*: all that is required is to exhibit a proof that t satisfies the parametricity predicate corresponding to its type. Pédrot and Tabareau [2018] exploit this approach to show the independence of premises with a term that uses exceptions locally as a backtracking reasoning technique.

The exceptional parametric translation, denoted $[-]_\varepsilon$, is the standard parametricity translation parametrized by the exceptional translation $[-]$. The exceptional parametric translation enjoys an abstraction theorem similar to standard parametricity, stated as follows:

Theorem 2.1 (Pédrot and Tabareau [2018]).
If $\vdash M : A$ *and* $[M]_\varepsilon$ *is defined, then* $\vdash [M]_\varepsilon : [A]_\varepsilon\ [M]$

The condition of $[M]_\varepsilon$ to be defined captures the extensibility of parametric reasoning in ETT: the translation is automatically defined for all CIC terms, and can be extended manually for terms that properly handle all their exceptions internally.

The purpose of the predicate $[A]_\varepsilon$ on a type A is to forbid the use of `raise` to inhabit it. Any type A in ETT is turned into a parametricity predicate $[A]_\varepsilon : [\![A]\!] \to \square$, which encodes the fact that an inhabitant of A is not allowed to generate an uncaught exception. Any occurrence of a term of the original theory used in the parametricity translation is replaced by its exceptional translation, using $[\cdot]$ or $[\![\cdot]\!]$ depending on whether it is used as a term or as a type. For instance, the translation of an application $[M\ N]_\varepsilon$ is given by $[M]_\varepsilon\ [N]\ [N]_\varepsilon$ instead of just $[M]_\varepsilon\ N\ [N]_\varepsilon$.

The translation of the universe is given by

$$[\square_i]_\varepsilon := \lambda A : [\![\square_i]\!].\ [\![A]\!] \to \square_i$$

where $[\![A]\!] := \mathtt{El}\ [A]$ is the translation of a term seen as a type (i.e., on the right-hand side of a typing judgment).

For inductive types, the default (error) constructor is always invalid, while all other constructors are valid, assuming their arguments are. For instance, the parametric translation $\mathbb{B}_\varepsilon : \mathbb{B}^\bullet \to \square$ for the inductive type $\mathbb{B}$ is an inductive type with only two constructors: $\mathtt{true}_\varepsilon$: $\mathbb{B}_\varepsilon\ \mathtt{true}^\bullet$ and $\mathtt{false}_\varepsilon$: $\mathbb{B}_\varepsilon\ \mathtt{false}^\bullet$. This means that $\mathtt{true}^\bullet$ and $\mathtt{false}^\bullet$ are parametric, but $\mathbb{B}_\varnothing$ is not.

As explained in the introduction, this new interpretation ensures logical consistency but rules out defining functions that let exceptions escape, let alone reasoning about such exceptional terms.

3 REASONABLY EXCEPTIONAL TYPE THEORY: DEFINITION

The Reasonably Exceptional Type Theory (RETT) supports consistent reasoning about exceptional programs by clearly separating three different universe hierarchies, and providing modalities to interoperate between them.

The (predicative) universe hierarchies of RETT are:

- the **exceptional layer**, $\square^{\mathsf{e}}$: this layer corresponds to plain ETT. It features an exception type **E** together with a failure function `raise`, and as such, is logically inconsistent.
- the **mediation layer**, $\square^{\mathsf{m}}$: this layer corresponds to the parametric fragment of ETT. While exceptions exist internally there, one must ensure that they are all caught before reaching toplevel. This safety discipline ensures consistency *a posteriori*.
- the **pure layer**, $\square^{\mathsf{p}}$: this layer only features the standard constructions of CIC. In particular, it does not allow raising exceptions at all.

The mediation layer supports the internalization of a parametricity predicate that classifies effectful terms that happen to be pure. All three layers are interpreted by translations to CIC, which in particular allows us to prove consistency and canonicity for the mediation and pure layers.

The interplay between these layers is threefold. First, dependent products are allowed to quantify over one layer in their domain, and another layer in their codomain (Section 3.1). Second, inductive types can be defined in all three layers, and can be eliminated into any other layer (Section 3.2). Crucially, elimination principles of inductive types depend on the source and target layers, in order to avoid compromising consistency. For instance, eliminating from $\square^{\mathsf{e}}$ to $\square^{\mathsf{m}}$ or $\square^{\mathsf{p}}$ requires explicitly accounting for potential exceptions. Third, RETT provides *modalities*—i.e. functions—connecting the mediation layer to the other two, both ways (Section 3.3). In particular, this allows RETT to feature an internal parametricity predicate that classifies effectful terms that happen to be pure (Section 3.4); this is key to allow generic reasoning about purity of exceptional terms. Finally, modalities and the parametricity predicate can be used to inject inductive types between layers (Section 3.5). This allows us to recover the standard elimination principle in the mediation layer, for impure inductive types of the exception layer, providing that the term on which we do elimination is parametric.

Why three layers? One may be suprised by the existence of three layers, rather than just two, namely one for effectful programming and one for pure reasoning. In a nutshell, this is because we cannot have a single layer that is both compatible with extensional properties (e.g. function extensionality), and suitable to define an internal parametricity predicate.

The pure layer satisfies the former but not the latter. It is merely an embedding of CIC, and thus proves the same theorems when they do not involve effectful programs. This conservativity result (Section 4.7) is very important as it allows us to backport any additional property one may want to add to CIC in the pure layer. For instance, the pure layer is compatible with functional extensionality, univalence or uniqueness of identity proofs. The disadvantage is that it does not give rise to an internal parametricity predicate, and so does not allow generic reasoning about purity of exceptional terms. This is intuitively because the pure layer is completely agnostic to exceptions. This intuition is made precise thanks to the translation presented in Section 4.5.

Dually, the mediation layer is but a logically-consistent restriction of the exceptional layer: it contains exceptions, but tamed by parametricity. It thus enables the definition of an internal parametricity predicate by using the modality from $\square^{\mathsf{m}}$ to $\square^{\mathsf{e}}$. Again, this intuition is made precise in Section 4.5. The price to pay is that the mediation layer gains impure property from the presence of internal exceptions, most notably the fact that it negates function extensionality. See Section 4.7 for more on this topic.

3.1 Negative Fragment

Figure 1 presents the syntax and typing rules of RETT. Apart from the three universe hierarchies and their corresponding binder and application annotations, the syntax is standard.

The first rules are standard and apply for all hierarchies. To make layer constraints explicit, we use $\square^{s}$ where s ranges over the layer identifiers `e`, `m`, and `p`. For instance, the universe rule specifies that each layer contains a denumerable well-founded sequence, but isolated from each other. Although the RETT syntactic model supports it, for simplicity the system featured in this paper does not feature cumulativity.

For technical reasons, we also annotate binders and applications with the layer in which the type of their argument lives, but we will often omit these annotations when they are clear from the context. Importantly, the dependent product is allowed to quantify over a type A from a universe in any layer; the layer of the dependent product type is determined by the layer of its codomain. This means that the dependent product crosscuts the three hierarchies, e.g. it is sufficient for B to live in $\square^{\mathsf{p}}$ to ensure that $\Pi x :^{\mathsf{e}} A.\, B$ lives in $\square^{\mathsf{p}}$, even when quantifying over A in $\square^{\mathsf{e}}$.

The typing of exceptions is the same as in ETT, save for the extra precision of the universe hierarchy: the exception type $\mathbf{E}$ lives in the exceptional layer, and likewise `raise` can only raise types from $\square^{\mathsf{e}}$.

Other than these specificities, the typing rules are standard. The rules for conversion, also standard, are given in Figure 2.

Syntax

$$\begin{array}{lcl} s & ::= & \mathtt{e} \mid \mathtt{m} \mid \mathtt{p} \\ A, B, M, N, P & ::= & \square_i^s \mid x \mid M\ {}^s N \mid \lambda x :^s A.\, M \mid \Pi x :^s A.\, B \mid \mathbf{const} \\ \Gamma, \Delta & ::= & \cdot \mid \Gamma, x :^s A \end{array}$$

Rules

$$\frac{}{\vdash \cdot} \qquad \frac{\Gamma \vdash A : \square_i^s}{\vdash \Gamma, x :^s A} \qquad \frac{\Gamma \vdash A : \square_i^s}{\Gamma, x :^s A \vdash x : A} \qquad \frac{\Gamma \vdash M : B \qquad \Gamma \vdash A : \square_i^s}{\Gamma, x :^s A \vdash M : B}$$

$$\frac{\Gamma \vdash M : B \qquad \Gamma \vdash A : \square_i^s \qquad A \equiv B}{\Gamma \vdash M : A} \qquad \frac{\vdash \Gamma \qquad i < j}{\Gamma \vdash \square_i^s : \square_j^s}$$

$$\frac{\Gamma \vdash A : \square_i^{s_1} \qquad \Gamma, x :^{s_1} A \vdash B : \square_j^{s_2}}{\Gamma \vdash \Pi x :^{s_1} A.\, B : \square_{\max(i,j)}^{s_2}} \qquad \frac{\Gamma, x :^{s_1} A \vdash M : B \qquad \Gamma \vdash \Pi x :^{s_1} A.\, B : \square_i^{s_2}}{\Gamma \vdash \lambda x :^{s_1} A.\, M : \Pi x :^{s_1} A.\, B}$$

$$\frac{\Gamma \vdash M : \Pi x :^s A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ {}^s N : B\{x := N\}}$$

Constants

$$\begin{array}{lcl} \mathbf{E} & : & \square_0^{\mathtt{e}} \\ \mathtt{raise} & : & \Pi A : \square_i^{\mathtt{e}}.\, \mathbf{E} \to A \end{array}$$

Fig. 1. RETT: Syntax and typing rules

$$(\lambda x :^s A.\, M)\ {}^s N \equiv M\{x := N\} \qquad \text{(congruence rules omitted)}$$

$$\mathtt{raise}\ {}^{s_2} (\Pi x :^{s_1} A.\, B)\ {}^{\mathtt{e}} M \equiv \lambda x :^{s_1} A.\, \mathtt{raise}\ {}^{s_2} B\ {}^{\mathtt{e}} M$$

Fig. 2. RETT: Conversion rules

3.2 Inductive Types

Each layer e, m or p of RETT contains inductively generated types. Similarly to what happens in vanilla Coq with the Prop-Type distinction [Bertot and Castéran 2004], RETT inductive types and their elimination principles thus need to be specifically placed in a given "home" layer, $\square^{\mathtt{e}}$, $\square^{\mathtt{m}}$ or $\square^{\mathtt{p}}$. When restricting the RETT system to a particular layer, this gives a full interpretation of CIC per hierarchy. We will thus not describe in detail their formation and elimination rules, as they are essentially the same as in CIC.

Inductive types meant to be used in programs must be placed in the exceptional or mediation layers. Conversely, inductive types meant for logical purposes must be placed in the mediation or pure layers. The dual role of the mediation layer will be explained thanks to the use of modalities later on. We give a few examples in Figure 3. To contrast the difference between the exceptional and mediation layers, we provide two variants of the booleans, $\mathbb{B}^{\mathtt{e}}$ in $\square^{\mathtt{e}}$ and $\mathbb{B}^{\mathtt{m}}$ in $\square^{\mathtt{m}}$. Note how the predicate of each eliminator lands in the same layer as the inductive type.

Note that because the exceptional layer features closed inductive terms that are not convertible to constructors, the reduction rules for eliminators over inductive types living in $\square^{\mathtt{e}}$ need to be extended to handle the raise term, by simply re-raising it. This is dictated by the usual semantics of call-by-name exceptions [Pédrot and Tabareau 2018].

Inductive constants

$$
\begin{array}{lcl}
\mathbb{B}^{\mathsf{e}} & : & \square_i^{\mathsf{e}} \\
\mathtt{true}^{\mathsf{e}} & : & \mathbb{B}^{\mathsf{e}} \\
\mathtt{false}^{\mathsf{e}} & : & \mathbb{B}^{\mathsf{e}} \\
\mathtt{rec}_{\mathbb{B}^{\mathsf{e}}} & : & \Pi P : \mathbb{B}^{\mathsf{e}} \to \square_i^{\mathsf{e}}.\, P\ \mathtt{true}^{\mathsf{e}} \to P\ \mathtt{false}^{\mathsf{e}} \to \Pi b : \mathbb{B}^{\mathsf{e}}.\, P\ b \\
\mathbb{B}^{\mathsf{m}} & : & \square_i^{\mathsf{m}} \\
\mathtt{true}^{\mathsf{m}} & : & \mathbb{B}^{\mathsf{m}} \\
\mathtt{false}^{\mathsf{m}} & : & \mathbb{B}^{\mathsf{m}} \\
\mathtt{rec}_{\mathbb{B}^{\mathsf{m}}} & : & \Pi P : \mathbb{B}^{\mathsf{m}} \to \square_i^{\mathsf{m}}.\, P\ \mathtt{true}^{\mathsf{m}} \to P\ \mathtt{false}^{\mathsf{m}} \to \Pi b : \mathbb{B}^{\mathsf{m}}.\, P\ b \\
\mathtt{list} & : & \square_i^{\mathsf{m}} \to \square_i^{\mathsf{m}} \\
\mathtt{nil} & : & \Pi A : \square_i^{\mathsf{m}}.\, \mathtt{list}\ A \\
\mathtt{cons} & : & \Pi A : \square_i^{\mathsf{m}}.\, A \to \mathtt{list}\ A \to \mathtt{list}\ A \\
\mathtt{rec}_{\mathtt{list}} & : & \Pi (A : \square_i^{\mathsf{m}})\, (P : \mathtt{list}\ A \to \square_i^{\mathsf{m}}). \\
 & & \quad P\ (\mathtt{nil}\ A) \to (\Pi (x : A)\, (l : \mathtt{list}\ A).\, P\ l \to P\ (\mathtt{cons}\ A\ x\ l)) \to \Pi l : \mathtt{list}\ A.\, P\ l \\
\mathtt{eq} & : & \Pi A : \square_i^{\mathsf{p}}.\, A \to A \to \square_i^{\mathsf{p}} \\
\mathtt{refl} & : & \Pi (A : \square_i^{\mathsf{p}})\, (x : A).\, \mathtt{eq}\ A\ x\ x \\
\mathtt{rec}_{\mathtt{eq}} & : & \Pi (A : \square_i^{\mathsf{p}})\, (x : A)\, (P : \Pi y : A.\, \mathtt{eq}\ A\ x\ y \to \square_i^{\mathsf{p}}).\, P\ x\ (\mathtt{refl}\ A\ x) \to \Pi (y : A)\, (e : \mathtt{eq}\ A\ x\ y).\, P\ y\ e
\end{array}
$$

Conversion rules

$$\mathtt{rec}_{\mathbb{B}^{\mathsf{m}}}\ P\ P_t\ P_f\ \mathtt{true}^{\mathsf{m}} \equiv P_t \qquad \mathtt{rec}_{\mathbb{B}^{\mathsf{m}}}\ P\ P_t\ P_f\ \mathtt{false}^{\mathsf{m}} \equiv P_f$$

$$\mathtt{rec}_{\mathbb{B}^{\mathsf{e}}}\ P\ P_t\ P_f\ \mathtt{true}^{\mathsf{e}} \equiv P_t \qquad \mathtt{rec}_{\mathbb{B}^{\mathsf{e}}}\ P\ P_t\ P_f\ \mathtt{false}^{\mathsf{e}} \equiv P_f$$

$$\mathtt{rec}_{\mathbb{B}^{\mathsf{e}}}\ P\ P_t\ P_f\ (\mathtt{raise}\ \mathbb{B}^{\mathsf{e}}\ M) \equiv \mathtt{raise}\ (P\ (\mathtt{raise}\ \mathbb{B}^{\mathsf{e}}\ M))\ M$$

$$\mathtt{rec}_{\mathtt{list}}\ A\ P\ P_n\ P_c\ (\mathtt{nil}\ A) \equiv P_n \qquad \mathtt{rec}_{\mathtt{list}}\ A\ P\ P_n\ P_c\ (\mathtt{cons}\ A\ M\ L) \equiv P_c\ M\ L\ (\mathtt{rec}_{\mathtt{list}}\ A\ P\ P_n\ P_c\ L)$$

$$\mathtt{rec}_{\mathtt{eq}}\ A\ M\ P\ P_r\ M\ (\mathtt{refl}\ A\ M) \equiv P_r$$

Fig. 3. Examples of inductive types in various layers

On the other hand, the existence of additional exception-raising terms in $\square^{\mathsf{e}}$ is also reflected by the ability to handle exceptions on inductive types.

Catch Eliminators. Inductive types in the exceptional layer additionally feature a *catch eliminator*, which is the same as the standard eliminator extended with a premise for the `raise` term, which furthermore satisfy the expected equations of a `try`/`with` block.

For instance, effectful booleans are equipped with the constant

$$\mathtt{catch}_{\mathbb{B}^{\mathsf{e}}} : \Pi P : \mathbb{B}^{\mathsf{e}} \to \square_i^{\mathsf{e}}.\, P\ \mathtt{true}^{\mathsf{e}} \to P\ \mathtt{false}^{\mathsf{e}} \to (\Pi e : \mathbf{E}.\, P\ (\mathtt{raise}\ \mathbb{B}^{\mathsf{e}}\ e)) \to \Pi b : \mathbb{B}^{\mathsf{e}}.\, P\ b$$

which is subject to the following equations

$$\mathtt{catch}_{\mathbb{B}^{\mathsf{e}}}\ P\ P_t\ P_f\ P_e\ \mathtt{true}^{\mathsf{e}} \equiv P_t \qquad \mathtt{catch}_{\mathbb{B}^{\mathsf{e}}}\ P\ P_t\ P_f\ P_e\ \mathtt{false}^{\mathsf{e}} \equiv P_f$$

$$\mathtt{catch}_{\mathbb{B}^{\mathsf{e}}}\ P\ P_t\ P_f\ P_e\ (\mathtt{raise}\ \mathbb{B}^{\mathsf{e}}\ M) \equiv P_e\ M$$

Note that the usual eliminator for exceptional inductive types (e.g. $\mathtt{rec}_{\mathbb{B}^{\mathsf{e}}}$) can actually be derived from the catch eliminator by re-raising the exception in the handling branch. The resulting terms satisfy the expected equations automatically.

This generalized eliminator permits writing exception-handling code in the exceptional layer, as if this fragment was an impure programming language. If we were to use a pattern-matching based

Source		Return type: $\square^{\mathsf{e}}$	$\square^{\mathsf{m}}$	$\square^{\mathsf{p}}$
	$\square^{\mathsf{e}}$	`rec/catch`	`catch`	`catch`
	$\square^{\mathsf{m}}$	`rec`	`rec`	–
	$\square^{\mathsf{p}}$	`rec`	`rec`	`rec`

Fig. 4. Legal mixed-layer eliminators

presentation, it would simply correspond to an optional additional exception-handling branch for exceptional inductive types.

Mixed Elimination. An even more interesting phenomenon at play is the interaction between the various layers. Indeed, eliminating an inductive type living in a layer s_1 into a layer s_2 needs to fulfill the invariants corresponding to their respective layers. This is not unlike what happens when eliminating from `Prop` to `Type` in CIC, insofar as one has to respect the singleton elimination criterion [Letouzey 2004]. Rather than having to decide that an elimination is proof-irrelevant, in RETT one has to check that an elimination cannot endanger consistency by allowing stray exceptions to land in a consistent layer.

This restriction is materialized by the fact that when eliminating from the unsafe $\square^{\mathsf{e}}$ layer to a safe layer, one has to explicitly handle all potential exceptions, by providing a catch-all clause. In practice, this means that there is no standard eliminator, only a catch eliminator. Given a layer for the eliminated inductive and a layer for the return predicate, the legal eliminators are summarized in Figure 4.

Note that for technical reasons that will be explained in the model construction (Section 4), it is not possible to eliminate from the mediation layer into the pure layer. Let us also insist that catch eliminators only make sense on exceptional inductive types, as the other layers lack a `raise` term, and the catch eliminator would not type-check.

3.3 Navigating Between Hierarchies

The main novelty of RETT is to provide modalities to navigate between the different layers, which are given below.

$$\{\,-\,\}^{\mathsf{e}}_{\mathsf{m}} : \square^{\mathsf{e}} \to \square^{\mathsf{m}} \qquad \{\,-\,\}^{\mathsf{m}}_{\mathsf{e}} : \square^{\mathsf{m}} \to \square^{\mathsf{e}} \qquad \{\,-\,\}^{\mathsf{m}}_{\mathsf{p}} : \square^{\mathsf{m}} \to \square^{\mathsf{p}} \qquad \{\,-\,\}^{\mathsf{p}}_{\mathsf{m}} : \square^{\mathsf{p}} \to \square^{\mathsf{m}}$$

Although they are written in a uniform way, they have wildly different computational behaviors, reflecting the different properties of the three universe hierarchies.

The modality $\{\,-\,\}^{\mathsf{e}}_{\mathsf{m}}$ amounts to considering that all terms in an exceptional type A, including exceptions, are parametric when seen in $\{A\}^{\mathsf{e}}_{\mathsf{m}}$. Intuitively, one can understand these terms as suspended computations, which are therefore trivially harmless. The modality $\{\,-\,\}^{\mathsf{m}}_{\mathsf{e}}$ is just a forgetful functor, which forgets the notion of parametricity attached to a type in the mediation layer, thus releasing its ability to raise exceptions. The modality $\{\,-\,\}^{\mathsf{m}}_{\mathsf{p}}$ forgets both about parametricity and the ability to raise an exception at that type, seeing it as a pure type. The modality $\{\,-\,\}^{\mathsf{p}}_{\mathsf{m}}$ equips a pure type with a default way to raise an exception, but automatically forbidden by the equipped notion of parametricity.

Most notably, the two modalities originating from the mediation layer are well-behaved in the sense that they commute with type formers, while the other ones enjoy no such property. The reason is that behind the scenes, $\{\,-\,\}^{\mathsf{m}}_{\mathsf{e}}$ and $\{\,-\,\}^{\mathsf{m}}_{\mathsf{p}}$ are the only modalities which correspond to forgetful functors and do not add anything to the type. In particular, sending a mediation type into the exceptional layer results in an exceptional type.

$$\{\square_i^{\mathsf{m}}\}_{\mathsf{e}}^{\mathsf{m}} \equiv \square_i^{\mathsf{e}}$$

This equation should really be thought of as the fact that $\square^{\mathsf{e}}$ is a semantic supertype of $\square^{\mathsf{m}}$, or dually, that $\square^{\mathsf{m}}$ is a restriction of $\square^{\mathsf{e}}$. We insist that homologous conversions do not hold for any other modality. Likewise, the two modalities originating from the mediation layer commute with products.

$$\{\Pi x :^s A.\, B\}_{\mathsf{e}}^{\mathsf{m}} \equiv \Pi x :^s A.\, \{B\}_{\mathsf{e}}^{\mathsf{m}} \quad \{\Pi x :^s A.\, B\}_{\mathsf{p}}^{\mathsf{m}} \equiv \Pi x :^s A.\, \{B\}_{\mathsf{p}}^{\mathsf{m}}$$

These modalities are equipped with the corresponding introduction operators

$$\iota_{\mathsf{m}}^{\mathsf{e}} : \Pi A : \square^{\mathsf{e}}.\, A \to \{A\}_{\mathsf{m}}^{\mathsf{e}} \quad \iota_{\mathsf{e}}^{\mathsf{m}} : \Pi A : \square^{\mathsf{m}}.\, A \to \{A\}_{\mathsf{e}}^{\mathsf{m}}$$

$$\iota_{\mathsf{m}}^{\mathsf{p}} : \Pi A : \square^{\mathsf{p}}.\, A \to \{A\}_{\mathsf{m}}^{\mathsf{p}} \quad \iota_{\mathsf{p}}^{\mathsf{m}} : \Pi A : \square^{\mathsf{m}}.\, A \to \{A\}_{\mathsf{p}}^{\mathsf{m}}$$

The corresponding equations hold on λ-abstractions. Note that the the commutation of modalities with Π-types is necessary for these equations to be well-typed.

$$\iota_{\mathsf{e}}^{\mathsf{m}}\ (\Pi x :^s A.\, B)\ (\lambda x :^s A.\, M) \equiv \lambda x :^s A.\, \iota_{\mathsf{e}}^{\mathsf{m}}\ B\ M$$

$$\iota_{\mathsf{p}}^{\mathsf{m}}\ (\Pi x :^s A.\, B)\ (\lambda x :^s A.\, M) \equiv \lambda x :^s A.\, \iota_{\mathsf{p}}^{\mathsf{m}}\ B\ M$$

The four modalities enjoy elimination principles, as long as the return type of the predicate is living in the same hierarchy as the target of the modality. Note that in general, there are no eliminators with a predicate living in a distinct layer.

$$\begin{array}{lcl}
\mathit{elim}_{\mathsf{m}}^{\mathsf{e}} & : & \Pi(A : \square^{\mathsf{e}})\ (P : \{A\}_{\mathsf{m}}^{\mathsf{e}} \to \square^{\mathsf{m}}).\ (\Pi a : A.\, P\ (\iota_{\mathsf{m}}^{\mathsf{e}}\ A\ a)) \to \Pi x : \{A\}_{\mathsf{m}}^{\mathsf{e}}.\, P\ x \\
\mathit{elim}_{\mathsf{p}}^{\mathsf{m}} & : & \Pi(A : \square^{\mathsf{m}})\ (P : \{A\}_{\mathsf{p}}^{\mathsf{m}} \to \square^{\mathsf{p}}).\ (\Pi a : A.\, P\ (\iota_{\mathsf{p}}^{\mathsf{m}}\ A\ a)) \to \Pi x : \{A\}_{\mathsf{p}}^{\mathsf{m}}.\, P\ x \\
\mathit{elim}_{\mathsf{m}}^{\mathsf{p}} & : & \Pi(A : \square^{\mathsf{p}})\ (P : \{A\}_{\mathsf{m}}^{\mathsf{p}} \to \square^{\mathsf{m}}).\ (\Pi a : A.\, P\ (\iota_{\mathsf{m}}^{\mathsf{p}}\ A\ a)) \to \Pi x : \{A\}_{\mathsf{m}}^{\mathsf{p}}.\, P\ x \\
\mathit{elim}_{\mathsf{e}}^{\mathsf{m}} & : & \Pi(A : \square^{\mathsf{m}})\ (P : \{A\}_{\mathsf{e}}^{\mathsf{m}} \to \square^{\mathsf{e}}).\ (\Pi a : A.\, P\ (\iota_{\mathsf{e}}^{\mathsf{m}}\ A\ a)) \to \Pi x : \{A\}_{\mathsf{e}}^{\mathsf{m}}.\, P\ x
\end{array}$$

These eliminators satisfy the expected reduction rules, e.g.

$$\mathit{elim}_{\mathsf{m}}^{\mathsf{e}}\ A\ P\ P_i\ (\iota_{\mathsf{m}}^{\mathsf{e}}\ A\ M) \equiv P_i\ M$$

Going from $\square^{\mathsf{e}}$ to $\square^{\mathsf{m}}$ and going back is the identity because it equips an exceptional type with a default notion of parametricity and directly forgets it, which is formally described by the following conversions:

$$\{\{A\}_{\mathsf{m}}^{\mathsf{e}}\}_{\mathsf{e}}^{\mathsf{m}} \equiv A \quad \text{and} \quad \iota_{\mathsf{e}}^{\mathsf{m}}\ \{A\}_{\mathsf{m}}^{\mathsf{e}}\ (\iota_{\mathsf{m}}^{\mathsf{e}}\ A\ M) \equiv M.$$

3.4 Internal Parametricity

RETT also features an internal parametricity predicate $\mathcal{P}$ on types of the form $\{A\}_{\mathsf{e}}^{\mathsf{m}}$, which classifies effectful terms that happen to be pure. It comes equipped with an injection and a projection

$$\begin{array}{lcl}
\mathcal{P} & : & \Pi A : \square^{\mathsf{m}}.\, \{A\}_{\mathsf{e}}^{\mathsf{m}} \to \square^{\mathsf{m}} \\
\iota_{\mathcal{P}} & : & \Pi(A : \square^{\mathsf{m}})\ (a : A).\, \mathcal{P}\ A\ (\iota_{\mathsf{e}}^{\mathsf{m}}\ A\ a) \\
\Downarrow_{\mathcal{P}} & : & \Pi(A : \square^{\mathsf{m}})\ (a : \{A\}_{\mathsf{e}}^{\mathsf{m}}).\, \mathcal{P}\ A\ a \to A
\end{array}$$

that satisfy the following equations

$$\Downarrow_{\mathcal{P}}\ A\ (\iota_{\mathsf{e}}^{\mathsf{m}}\ A\ M)\ (\iota_{\mathcal{P}}\ A\ M) \equiv M \quad \iota_{\mathsf{e}}^{\mathsf{m}}\ A\ (\Downarrow_{\mathcal{P}}\ A\ M\ P) \equiv M$$

Interestingly, using these terms, we can provide a specific elimination principle $\mathit{elim}_{\mathcal{P}}$ that enables reasoning on $\{-\}_{\mathsf{e}}^{\mathsf{m}}$ with predicates living in the mediation layer. As such, it is the mediation-landing version of the $\mathit{elim}_{\mathsf{e}}^{\mathsf{m}}$ eliminator, and intuitively the requirement that the term being eliminated is parametric corresponds to the invariant that it should not raise uncaught exceptions.

The parametricity eliminator is defined as

$$\begin{array}{lll} elim_{\mathcal{P}} & : & \Pi(A:\square^{\mathsf{m}})\,(P:\{A\}^{\mathsf{m}}_{\mathsf{e}}\to\square^{\mathsf{m}}).\,(\Pi x:A.\,P\,(\iota^{\mathsf{m}}_{\mathsf{e}}\,A\,x))\to\Pi(x:\{A\}^{\mathsf{m}}_{\mathsf{e}})\,(p:\mathcal{P}\,A\,x).\,P\,x \\ & := & \lambda A\,P\,P_i\,x\,p.\,P_i\,(\Downarrow_{\mathcal{P}}\,A\,x\,p). \end{array}$$

Lemma 3.1. *The parametricity eliminator satisfies the expected ι-reduction:*

$$elim_{\mathcal{P}}\;A\;P\;P_i\;(\iota^{\mathsf{m}}_{\mathsf{e}}\;A\;M)\;(\iota_{\mathcal{P}}\;A\;M)\equiv P_i\;M.$$

Internal parametricity lets the user separate the implementation of a term that potentially uses exceptions internally from its specification, ensuring that it is observationally pure. We use it in the next section to derive standard elimination of exceptional inductive types into the mediation layer, up to a proof of parametricity of the exceptional term.

Similarly to what happens with corresponding modalities, the parametricity predicate lives in the mediation layer, and thus commutes with dependent products.

Lemma 3.2. *Using the above combinators, it is possible to write terms*

$$\mathcal{P}\_\mathtt{to}\_\Pi:\Pi(A:\square^s)\,(B:A\to\square^{\mathsf{m}})\,(f:\{\Pi x:^s A.\,B\,x\}^{\mathsf{m}}_{\mathsf{e}}).\,\mathcal{P}\,(\Pi x:^s A.\,B\,x)\,f\to\Pi x:^s A.\,\mathcal{P}\,(B\,x)\,(f\,x)$$
$$\mathcal{P}\_\mathtt{of}\_\Pi:\Pi(A:\square^s)\,(B:A\to\square^{\mathsf{m}})\,(f:\{\Pi x:^s A.\,B\,x\}^{\mathsf{m}}_{\mathsf{e}}).\,(\Pi x:^s A.\,\mathcal{P}\,(B\,x)\,(f\,x))\to\mathcal{P}\,(\Pi x:^s A.\,B\,x)\,f$$

Those terms satisfy unsurprising equations (not detailed here) that can be used to easily transfer parametricity conditions under and over contexts.

3.5 Parametricity for Exceptional Inductive Types

We now show that the $\{-\}^{\mathsf{m}}_{\mathsf{e}}$ modality together with the parametricity predicate $\mathcal{P}$ make it possible to define a notion of parametricity on exceptional inductive types. The notion of parametricity in turn allows us to derive standard elimination principles *into the mediation layer* for exceptional inductive types, with an extra guard condition that the eliminated exceptional term is parametric.

Definition 3.3. Let $\mathcal{I}:\Pi(x_1:^{s_1}X_1)\ldots(x_n:^{s_n}X_n).\,\square^{\mathsf{m}}$ be an inductive type in the mediation layer. We call $\{\mathcal{I}\}^{\mathsf{m}}_{\mathsf{e}}$ the *exceptional lowering* of $\mathcal{I}$, which is typed as $\{\mathcal{I}\}^{\mathsf{m}}_{\mathsf{e}}:\Pi(x_1:^{s_1}X_1)\ldots(x_n:^{s_n}X_n).\,\square^{\mathsf{e}}$.

We now show how the lowering of an inductive type from the mediation layer (called a mediation inductive type for short) satisfies the parametric elimination principle mentioned above and is equivalent to its corresponding exceptional inductive type in the case of booleans and lists.

Non-recursive types. Lowering a non-recursive mediation inductive type through $\{-\}^{\mathsf{m}}_{\mathsf{e}}$ results in an inductive type (e.g. $\{\mathbb{B}^{\mathsf{m}}\}^{\mathsf{m}}_{\mathsf{e}}$) that behaves just like an exceptional inductive type (e.g. $\mathbb{B}^{\mathsf{e}}$). Lowered inductive types can be introduced by the corresponding injection of their constructors. For instance, in the case of booleans, we have

$$\iota^{\mathsf{m}}_{\mathsf{e}}\,\mathbb{B}^{\mathsf{m}}\,\mathtt{true}^{\mathsf{m}}:\{\mathbb{B}^{\mathsf{m}}\}^{\mathsf{m}}_{\mathsf{e}}\quad\text{and}\quad\iota^{\mathsf{m}}_{\mathsf{e}}\,\mathbb{B}^{\mathsf{m}}\,\mathtt{false}^{\mathsf{m}}:\{\mathbb{B}^{\mathsf{m}}\}^{\mathsf{m}}_{\mathsf{e}}.$$

Usual eliminators targetting the exceptional layer can be derived using the eliminators for the corresponding mediation inductive together with the eliminator for the lowering modality. We can e.g. implement a term

$$\mathtt{rec}_{\{\mathbb{B}^{\mathsf{m}}\}^{\mathsf{m}}_{\mathsf{e}}}:\Pi P:\{\mathbb{B}^{\mathsf{m}}\}^{\mathsf{m}}_{\mathsf{e}}\to\square^{\mathsf{e}}.\,P\,(\iota^{\mathsf{m}}_{\mathsf{e}}\,\mathbb{B}^{\mathsf{m}}\,\mathtt{true}^{\mathsf{m}})\to P\,(\iota^{\mathsf{m}}_{\mathsf{e}}\,\mathbb{B}^{\mathsf{m}}\,\mathtt{false}^{\mathsf{m}})\to\Pi b:\{\mathbb{B}^{\mathsf{m}}\}^{\mathsf{m}}_{\mathsf{e}}.\,P\,b$$

satisfying the expected ι-rules for constructors *only*. The reduction rule of the modality eliminator on `raise` is indeed not specified, which prevents extending the equation on `raise` to the lowered inductive version.

By restricting oneself to the case of *parametric* inductive terms, it is also possible to write an eliminator that targets the mediation layer. It is readily implemented by chaining the eliminator for internal parametricity with the one for the inductive type under consideration. For booleans, this results in a *parametric eliminator*

$$\mathtt{rec}^{\mathcal{P}}_{\{\mathbb{B}^{\mathsf{m}}\}^{\mathsf{m}}_{\mathsf{e}}}:\Pi P:\{\mathbb{B}^{\mathsf{m}}\}^{\mathsf{m}}_{\mathsf{e}}\to\square^{\mathsf{m}}.\,P\,(\iota^{\mathsf{m}}_{\mathsf{e}}\,\mathbb{B}^{\mathsf{m}}\,\mathtt{true}^{\mathsf{m}})\to P\,(\iota^{\mathsf{m}}_{\mathsf{e}}\,\mathbb{B}^{\mathsf{m}}\,\mathtt{false}^{\mathsf{m}})\to\Pi b:\{\mathbb{B}^{\mathsf{m}}\}^{\mathsf{m}}_{\mathsf{e}}.\,\mathcal{P}\,\mathbb{B}^{\mathsf{m}}\,b\to P\,b$$

that is also subject to the expected conversion rules. Such parametric eliminators allow us to reason in the mediation layer about the purity of terms of exceptional inductive types.

Unfortunately, catch eliminators for lowered inductive types are not derivable from the set of primitive combinators at hand. Thankfully, lowered catch eliminators are nonetheless valid in the model (i.e. one can provide a term in the target theory whose type is the translation of the corresponding source type), and thus can be postulated in RETT. For booleans, this amounts to stating that RETT is extended with terms of type

$$\begin{aligned}\mathtt{catch}_{\{\mathbb{B}^{\mathtt{m}}\}^{\mathtt{m}}_{\mathtt{e}}} : \Pi P : \{\mathbb{B}^{\mathtt{m}}\}^{\mathtt{m}}_{\mathtt{e}} \to \square^{s}.\\ P\ (\iota^{\mathtt{m}}_{\mathtt{e}}\ \mathbb{B}^{\mathtt{m}}\ \mathtt{true}^{\mathtt{m}}) \to P\ (\iota^{\mathtt{m}}_{\mathtt{e}}\ \mathbb{B}^{\mathtt{m}}\ \mathtt{false}^{\mathtt{m}}) \to (\Pi e : \mathbf{E}.\ P\ (\mathtt{raise}\ \{\mathbb{B}^{\mathtt{m}}\}^{\mathtt{m}}_{\mathtt{e}}\ e)) \to\\ \Pi b : \{\mathbb{B}^{\mathtt{m}}\}^{\mathtt{m}}_{\mathtt{e}}.\ P\ b\end{aligned}$$

for s ranging over e, m and p and subject to the full range of equations, that is, both the ones on constructors as well as the ones on raise.

One can now use these lowered catch eliminators to show that the lowering of a mediation inductive type is isomorphic to the corresponding exceptional inductive type. This makes explicit the dual nature of the mediation layer, which can be used both for safe reasoning, and for effectful computation through lowering.

Lemma 3.4. *$\{\mathbb{B}^{\mathtt{m}}\}^{\mathtt{m}}_{\mathtt{e}}$ is isomorphic to $\mathbb{B}^{\mathtt{e}}$.*

Proof. The two inductive types satisfy the same universal property, namely catch elimination, and thus are isomorphic. □

Moreover, the ability to catch failures on lowered inductive types can also be used to specify the parametricity predicate on them. That is, it is possible to prove that failure on lowered inductive types is never parametric[1], e.g. for booleans

Lemma 3.5. *The following type is inhabited in* RETT

$$\Pi(P : \{\mathbb{B}^{\mathtt{m}}\}^{\mathtt{m}}_{\mathtt{e}} \to \square^{\mathtt{m}})\ (e : \mathbf{E}).\ \mathcal{P}\ \mathbb{B}^{\mathtt{m}}\ (\mathtt{raise}\ \{\mathbb{B}^{\mathtt{m}}\}^{\mathtt{m}}_{\mathtt{e}}\ e) \to P\ (\mathtt{raise}\ \{\mathbb{B}^{\mathtt{m}}\}^{\mathtt{m}}_{\mathtt{e}}\ e).$$

This is easily obtained by combining the parametric eliminator with the catch eliminator.

Recursive types. We conclude this section with the specific case of lowering recursive inductive types, i.e. types that mention themselves in the type of their constructors. In this case, one has to be a little more careful than above, because lowering needs to be handled specially. The reason is that the $\{\,-\,\}^{\mathtt{m}}_{\mathtt{e}}$ modality does not distribute on the left-hand side of a Π-type, which means that there is a type mismatch for recursive constructors, e.g.

$$\iota^{\mathtt{m}}_{\mathtt{e}}\ (A \to \mathtt{list}\ A \to \mathtt{list}\ A)\ \mathtt{cons} : A \to \mathtt{list}\ A \to \{\mathtt{list}\ A\}^{\mathtt{m}}_{\mathtt{e}}$$

rather than

$$A \to \{\mathtt{list}\ A\}^{\mathtt{m}}_{\mathtt{e}} \to \{\mathtt{list}\ A\}^{\mathtt{m}}_{\mathtt{e}}$$

which would be required to obtain an inductive equivalent to the list datatype in $\square^{\mathtt{e}}$. This can be circumvented using the elimination principle of $\{\,-\,\}^{\mathtt{m}}_{\mathtt{e}}$ on the recursive arguments. For instance,

$$\mathit{elim}^{\mathtt{m}}_{\mathtt{e}}\ (\mathtt{list}\ A)\ (\lambda\_.\ \{\mathtt{list}\ A\}^{\mathtt{m}}_{\mathtt{e}})\ (\lambda l.\ \iota^{\mathtt{m}}_{\mathtt{e}}\ (\mathtt{list}\ A)\ (\mathtt{cons}\ A\ M\ l))\ N$$

has the adequate type as expected above.

[1]This is equivalent to saying that $\mathcal{P}\ \mathbb{B}^{\mathtt{m}}\ (\mathtt{raise}\ \{\mathbb{B}^{\mathtt{m}}\}^{\mathtt{m}}_{\mathtt{e}}\ e)$ implies $\bot_{\mathtt{m}}$, the inductive with no constructor in $\square^{\mathtt{m}}$.

$$
\begin{array}{lcl}
[\square_i^{\mathsf{e}}]^{\mathsf{e}} & := & \mathtt{TypeVal}\ \mathtt{type}_i\ \mathtt{TypeErr}_i \\
[x]^{\mathsf{e}} & := & x \\
[\lambda x :^{s} A.\, M]^{\mathsf{e}} & := & \lambda x : [\![A]\!]^{s}.\, [M]^{\mathsf{e}} \\
[M\ {}^{s} N]^{\mathsf{e}} & := & [M]^{\mathsf{e}}\ [N]^{s} \\
[\Pi x :^{s} A.\, B]^{\mathsf{e}} & := & \mathtt{TypeVal}\ (\Pi x : [\![A]\!]^{s}.\, [\![B]\!]^{\mathsf{e}})\ (\lambda (e : \mathbb{E})\ (x : [\![A]\!]^{s}).\, [B]_{\varnothing}\ e) \\
[\mathbf{E}]^{\mathsf{e}} & := & \mathtt{TypeVal}\ \mathbb{E}\ (\lambda e : \mathbb{E}.\, e) \\
[\mathtt{raise}]^{\mathsf{e}} & := & \lambda (A : \mathtt{type})\ (e : \mathbb{E}).\, [A]_{\varnothing}\ e \\
[\mathbb{B}]^{\mathsf{e}} & := & \mathtt{TypeVal}\ \mathbb{B}^{\bullet}\ \mathbb{B}_{\varnothing} \\
[\mathtt{list}]^{\mathsf{e}} & := & \lambda A : [\![\square^{\mathsf{e}}]\!].\, \mathtt{TypeVal}\ (\mathtt{list}^{\bullet}\ [A])\ \mathtt{list}_{\varnothing} \\
[c]^{\mathsf{e}} & := & c^{\bullet} \qquad \text{(for any constructor } c \text{ of an inductive type)} \\
[\mathtt{rec}_{\mathbb{B}}]^{\mathsf{e}} & := & \lambda P\, p_t\, p_f\, b.\ \ \mathtt{match}\ b\ \mathtt{return}\ \lambda b.\, \mathtt{El}\ (P\ b)\ \mathtt{with} \\
 & & \qquad\qquad \mid \mathtt{true}^{\bullet} \Rightarrow p_t \\
 & & \qquad\qquad \mid \mathtt{false}^{\bullet} \Rightarrow p_f \\
 & & \qquad\qquad \mid \mathbb{B}_{\varnothing}\ e \Rightarrow [P\ \mathbb{B}_{\varnothing}\ e]_{\varnothing}\ e \\
 & & \qquad\quad\ \ \mathtt{end} \\
[\mathtt{rec}_{\mathtt{list}}]^{\mathsf{e}} & := & \ldots \text{ (omitted for brevity)} \\
[A]_{\varnothing} & := & \mathtt{Err}\ [A]^{\mathsf{e}} \\
[\![A]\!]^{\mathsf{e}} & := & \mathtt{El}\ [A]^{\mathsf{e}} \\
[\![\cdot]\!] & := & \cdot \\
[\![\Gamma, x :^{\mathsf{e}} A]\!] & := & [\![\Gamma]\!], x : [\![A]\!]^{\mathsf{e}}
\end{array}
$$

$$
\begin{array}{l}
\mathtt{Ind}\ \mathbb{B}^{\bullet}\ : \square := \\
\mid \mathtt{true}^{\bullet} : \mathbb{B}^{\bullet} \\
\mid \mathtt{false}^{\bullet} : \mathbb{B}^{\bullet} \\
\mid \mathbb{B}_{\varnothing} : \mathbb{E} \to \mathbb{B}^{\bullet}
\end{array}
\qquad\Bigg\|\qquad
\begin{array}{l}
\mathtt{Ind}\ \mathtt{list}^{\bullet}\ (A : [\![\square^{\mathsf{e}}]\!])\ : \square := \\
\mid \mathtt{nil}^{\bullet} : \mathtt{list}^{\bullet}\ A \\
\mid \mathtt{cons}^{\bullet} : [\![A]\!] \to \mathtt{list}^{\bullet}\ A \to \mathtt{list}^{\bullet}\ A \\
\mid \mathtt{list}_{\varnothing} : \mathbb{E} \to \mathtt{list}^{\bullet}\ A
\end{array}
$$

Fig. 5. Translation of the exceptional layer

4 A SYNTACTIC MODEL OF RETT

We define the semantics of RETT by a syntactic translation into CIC, following the general technique of Boulier et al. [2017]. This allows us to straightforwardly prove its good metatheoretical properties, like consistency and canonicity. We first present the translations of each layer, then explain the translation of the modalities and finally prove the correctness of the translation and deduce metatheoretical properties.

4.1 Exceptional Layer

The translation of the exceptional layer is given in Figure 5. It follows exactly the translation given by Pédrot and Tabareau [2018], which we almost completely introduced in Section 2. Following the

$$
\begin{array}{lll}
[\square_i^{\mathsf{m}}]_\varepsilon & := & \lambda A : [\![\square_i^{\mathsf{m}}]\!].\, [\![A]\!]^{\mathsf{m}} \to \square_i \\
[x]_\varepsilon & := & x_\varepsilon \\
[\lambda x :^{\mathsf{m}} A.\, M]_\varepsilon & := & \lambda (x : [\![A]\!]^{\mathsf{m}})\, (x_\varepsilon : [\![A]\!]_\varepsilon\, x).\, [M]_\varepsilon \\
[\lambda x :^{s} A.\, M]_\varepsilon & := & \lambda x : [\![A]\!]^{s}.\, [M]_\varepsilon \qquad \text{if } s \in \{\mathsf{p}, \mathsf{e}\} \\
[M\, {}^{\mathsf{m}} N]_\varepsilon & := & [M]_\varepsilon\, [N]^{\mathsf{m}}\, [N]_\varepsilon \\
[M\, {}^{s} N]_\varepsilon & := & [M]_\varepsilon\, [N]^{s} \qquad \text{if } s \in \{\mathsf{p}, \mathsf{e}\} \\
[\Pi x :^{\mathsf{m}} A.\, B]_\varepsilon & := & \lambda (f : \Pi x : [\![A]\!]^{\mathsf{m}}.\, [\![B]\!]^{\mathsf{m}}).\, \Pi (x : [\![A]\!]^{\mathsf{m}})\, (x_\varepsilon : [\![A]\!]_\varepsilon\, x).\, [\![B]\!]_\varepsilon\, (f\, x) \\
[\Pi x :^{s} A.\, B]_\varepsilon & := & \lambda (f : \Pi x : [\![A]\!]^{s}.\, [\![B]\!]^{\mathsf{m}}).\, \Pi x : [\![A]\!]^{s}.\, [\![B]\!]_\varepsilon\, (f\, x) \qquad \text{if } s \in \{\mathsf{p}, \mathsf{e}\} \\
[\mathcal{I}]_\varepsilon & := & \mathcal{I}_\varepsilon \qquad \text{(for any inductive type } \mathcal{I}\text{)} \\
[c]_\varepsilon & := & c_\varepsilon \qquad \text{(for any constructor } c \text{ of an inductive type)} \\
[\![A]\!]_\varepsilon & := & [A]_\varepsilon \\
[\![\cdot]\!]_\varepsilon & := & \cdot \\
[\![\Gamma, x :^{\mathsf{m}} A]\!]_\varepsilon & := & [\![\Gamma]\!]_\varepsilon, x : [\![A]\!]^{\mathsf{m}}, x_\varepsilon : [\![A]\!]_\varepsilon\, x \\
[\![\Gamma, x :^{s} A]\!]_\varepsilon & := & [\![\Gamma]\!]_\varepsilon, x : [\![A]\!]^{s} \qquad \text{if } s \in \{\mathsf{p}, \mathsf{e}\}
\end{array}
$$

$$
\begin{array}{l}
\mathtt{Ind}\ \mathbb{B}_\varepsilon\ : \mathbb{B}^\bullet \to \square := \\
\mid \mathtt{true}_\varepsilon : \mathbb{B}_\varepsilon\ \mathtt{true}^\bullet \\
\mid \mathtt{false}_\varepsilon : \mathbb{B}_\varepsilon\ \mathtt{false}^\bullet
\end{array}
\qquad \Bigg\|\qquad
\begin{array}{l}
\mathtt{Ind}\ \mathtt{list}_\varepsilon\ (A : \mathtt{type})\, (A_\varepsilon : [\![A]\!] \to \square)\ : \mathtt{list}^\bullet\ A \to \square := \\
\mid \mathtt{nil}_\varepsilon : \mathtt{list}_\varepsilon\ A\, A_\varepsilon\ (\mathtt{nil}^\bullet\ A) \\
\mid \mathtt{cons}_\varepsilon : \Pi (x : [\![A]\!])\, (x_\varepsilon : A_\varepsilon\, x)\, (l : \mathtt{list}^\bullet\ A)\, (l_\varepsilon : \mathtt{list}_\varepsilon\ A\, A_\varepsilon\, l). \\
\qquad\qquad \mathtt{list}_\varepsilon\ A\, A_\varepsilon\ (\mathtt{cons}^\bullet\ A\, x\, l)
\end{array}
$$

Fig. 6. Parametricity translation

syntactic translation approach [Boulier et al. 2017], the term translation is written $[-]^{\mathsf{e}}$ and the type translation, written $[\![-]\!]^{\mathsf{e}}$, is derived from it using the function $\mathtt{El}$.[2] Note that in RETT the exceptional translation only applies to terms whose type is a sort in the exceptional universe $\square^{\mathsf{e}}$.

Recall that types are mapped to values of the inductive type `type`, which has two constructors, `TypeVal` and `TypeErr`. The former is used to represent valid types (as a pair of a type and its default function); the latter is the default function for errors on types. The only rule we did not explain in Section 2 is the translation of the dependent product: it simply produces a `TypeVal` whose representation type is the type component of the recursive translation on A and B, and whose default function re-raises the exception e on the default function for type B (retrieved using the macro $[-]_\varnothing$).

The translation handles inductive types following the approach of Pédrot and Tabareau [2018] briefly presented in Section 2. We provide the examples of booleans and lists for illustration. Essentially, an inductive type (e.g. $\mathbb{B}^{\mathsf{e}}$) is translated to a new inductive type (e.g. $\mathbb{B}^\bullet$) with an extra constructor (e.g. $\mathbb{B}_\varnothing$), used as the default function to raise exceptions at that type. The eliminators (e.g. $\mathtt{rec}_{\mathbb{B}}$) propagate exceptions in these new branches.

[2] Recall from Section 2 that $\mathtt{El}_i$ recovers the underlying type from an inhabitant of $\mathtt{type}_i$, and $\mathtt{Err}_i$ lifts the default function to this underlying type.

4.2 Mediation Layer

The translation $[-]^{\mathtt{m}}$ of the mediation layer is the same as that of the exceptional layer, replacing e with m, in particular:

$$[\square_i^{\mathtt{m}}]^{\mathtt{m}} \quad := \quad \mathtt{TypeVal}\ \mathtt{type}_i\ \mathtt{TypeErr}_i$$

$$[\![A]\!]^{\mathtt{m}} \quad := \quad \mathtt{El}\ [A]^{\mathtt{m}}$$

The peculiarity of the mediation layer is that every term also comes with its *parametricity proof*. This proof is obtained by the translation $[-]_\varepsilon$, described in Figure 6. This translation is essentially the standard parametricity translation for type theory [Bernardy and Lasson 2011], with a few adjustments specific to RETT. We stress that $[-]_\varepsilon$ is only defined for terms living in the mediation layer, so that writing $[M]_\varepsilon$ implicitly assumes $M : A$ for some $A : \square^{\mathtt{m}}$.

Let us first recall the basics of this translation. For the universe, the translation is defined as (arbitrary) predicates on types, i.e. if $A : \square^{\mathtt{m}}$ then $[A]_\varepsilon : [\![A]\!]^{\mathtt{m}} \to \square$. Dependent products, functions, and applications are defined by cases, but consider only the first line of each for now. For the dependent product, the translation specifies that given a parametric input x of type A—as witnessed by x_ε of type $[A]_\varepsilon\ x$—the function yields a parametric output of type B. Similarly, the translation of a lambda term is a function that takes an argument x and a witness x_ε that it is parametric; a variable x is translated to x_ε; a translated application (again, consider only the first line for now) passes the parametricity witness as an extra argument. The translation of type environments follows the same pattern, with parametricity witness x_ε.

The main specificity of the parametricity translation for RETT is that it must take into account the fact the dependent product in RETT can quantify over types in any layer, in particular those which are not coming with parametricity proofs. Therefore, the translation of the dependent product depends on the layer of the domain: if $x : A$ is in the mediation layer, then the parametricity predicate for its argument (x_ε) is required; otherwise, it is not. The translations of lambda terms and applications follow the same discipline: parametricity is only imposed on types and terms from the mediation layer. Second, as in ETT, the parametricity translation recursively triggers the base translation $[-]^s$ (or $[\![-]\!]^s$ depending on the position) on any occurrence of a RETT term from the layer s—see for instance the translation of an application, which uses $[N]^s$.

The parametric translation of inductive types, illustrated for booleans and lists, differs from the exceptional in two crucial ways: first, no default constructors are added. This is because the parametric translation imposes purity, and hence only the standard constructors are valid, assuming their arguments are. This latter condition means that parametricity witnesses are required: e.g., $\mathtt{cons}_\varepsilon$ requires the parametricity witness of both the added element (x_ε) and the list (l_ε).

4.3 Pure Layer

The pure translation (Fig. 7) is essentially the identity translation, but for the fact that it inductively makes use of previous translation when using a crosscutting dependent product whose domain is in another layer.

4.4 Mixed Eliminators

The interpretation of inductive types in each layer is given above, but it is worth mentioning the crosscutting eliminators. Indeed, as discussed in Section 3.2, there are ways to eliminate inductive terms on predicates landing in a different layer than the one the inductive type is living in.

The various catch eliminators are simply built out of the corresponding pattern-matching on all constructors. For instance, the $\mathbb{B}^{\mathtt{e}}$ eliminator into $\square^{\mathtt{m}}$ is defined in Figure 8. Note the intricate return type of the parametric component. Likewise, the catch eliminator into $\square^{\mathtt{e}}$ has the same base

$$
\begin{array}{lcl}
[\square_i^{\mathsf{p}}]^{\mathsf{p}} & := & \square_i \\
[x]^{\mathsf{p}} & := & x \\
[\lambda x :^s A.\, M]^{\mathsf{p}} & := & \lambda x : [\![A]\!]^s.\, [M]^{\mathsf{p}} \\
[M\, {}^s N]^{\mathsf{p}} & := & [M]^{\mathsf{p}}\ [N]^s \\
[\Pi x :^s A.\, B]^{\mathsf{p}} & := & \Pi x : [\![A]\!]^s.\, [B]^{\mathsf{p}} \\
[\Sigma x : A.\, B]^{\mathsf{p}} & := & \Sigma x : [\![A]\!]^{\mathsf{p}}.\, [B]^{\mathsf{p}} \\
[\mathsf{eq}\ A\ x\ y]^{\mathsf{p}} & := & \mathsf{eq}\ [\![A]\!]^{\mathsf{p}}\ [x]^{\mathsf{p}}\ [y]^{\mathsf{p}} \\
[\![A]\!]^{\mathsf{p}} & := & [A]^{\mathsf{p}}
\end{array}
$$

Fig. 7. Translation of the pure layer

$$
\begin{array}{lcl}
[\mathtt{catch}_{\mathbb{B}^{\mathsf{e}}}] & := & \lambda P : \mathbb{B}^{\mathsf{e}} \to \mathtt{type}. \\
& & \lambda(P_t : \mathtt{El}\ (P\ \mathtt{true}^\bullet))\ (P_f : \mathtt{El}\ (P\ \mathtt{false}^\bullet))\ (P_e : \Pi e : \mathbb{E}.\, \mathtt{El}\ (P\ (\mathbb{B}_\varnothing\ e))). \\
& & \lambda b : \mathbb{B}^{\mathsf{e}}. \\
& & \quad \mathtt{match}\ b\ \mathtt{return}\ \lambda b.\, \mathtt{El}\ (P\ b)\ \mathtt{with} \\
& & \quad \mid \mathtt{true}^\bullet \Rightarrow P_t \\
& & \quad \mid \mathtt{false}^\bullet \Rightarrow P_f \\
& & \quad \mid \mathbb{B}_\varnothing\ e \Rightarrow P_e\ e \\
& & \quad \mathtt{end} \\
[\mathtt{catch}_{\mathbb{B}^{\mathsf{e}}}]_\varepsilon & := & \lambda(P : \mathbb{B}^{\mathsf{e}} \to \mathtt{type})\ (P_\varepsilon : \Pi b : \mathbb{B}^{\mathsf{e}}.\, \mathtt{El}\ (P\ b) \to \square). \\
& & \lambda(P_t : \mathtt{El}\ (P\ \mathtt{true}^\bullet))\ (P_{t_\varepsilon} : P_\varepsilon\ \mathtt{true}^\bullet\ P_t). \\
& & \lambda(P_f : \mathtt{El}\ (P\ \mathtt{false}^\bullet))\ (P_{f_\varepsilon} : P_\varepsilon\ \mathtt{false}^\bullet\ P_f). \\
& & \lambda(P_e : \Pi e : \mathbb{E}.\, \mathtt{El}\ (P\ (\mathbb{B}_\varnothing\ e)))\ (P_{e_\varepsilon} : \Pi e : \mathbb{E}.\, P_\varepsilon\ (\mathbb{B}_\varnothing\ e)\ (P_e\ e)). \\
& & \lambda b : \mathbb{B}^{\mathsf{e}}. \\
& & \quad \mathtt{match}\ b\ \mathtt{return}\ \lambda b.\, P_\varepsilon\ b\ ([\mathtt{catch}_{\mathbb{B}^{\mathsf{e}}}]\ P\ P_t\ P_f\ P_e\ b)\ \mathtt{with} \\
& & \quad \mid \mathtt{true}^\bullet \Rightarrow P_{t_\varepsilon} \\
& & \quad \mid \mathtt{false}^\bullet \Rightarrow P_{f_\varepsilon} \\
& & \quad \mid \mathbb{B}_\varnothing\ e \Rightarrow P_{e_\varepsilon}\ e\ (P_e\ e) \\
& & \quad \mathtt{end}
\end{array}
$$

Fig. 8. Eliminating $\mathbb{B}^{\mathsf{e}}$ into $\square^{\mathsf{m}}$

translation, the main difference being that it does not have a $[-]_\varepsilon$ translation. The `catch` eliminator into $\square^{\mathsf{p}}$ is also very similar, the main difference being the removal of `El` casts.

We will not describe the other eliminators as they are straightforward. We will insist nonetheless on the reason why there is no eliminator from $\square^{\mathsf{m}}$ inductive types into $\square^{\mathsf{p}}$. This is due to the lack of internal parametricity in $\square^{\mathsf{p}}$. Given a value of $\mathbb{B}^{\mathsf{m}}$, through the $[-]^{\mathsf{m}}$ one also needs to handle the failure case in the $\square^{\mathsf{p}}$-returning pattern-matching, but there is no way to return a default value because in general types living in $\square^{\mathsf{p}}$ are not necessarily inhabited. One could then argue that it would still be possible to provide the `catch` variant. While it seems reasonable from a computational

$$
\begin{array}{lll|lll}
[\{A\}^{\mathsf{m}}_{\mathsf{e}}]^{\mathsf{e}} & := & [A]^{\mathsf{m}} & & & \\
[\{A\}^{\mathsf{m}}_{\mathsf{p}}]^{\mathsf{p}} & := & \mathtt{El}\ [A]^{\mathsf{m}} & & & \\
[\{A\}^{\mathsf{p}}_{\mathsf{m}}]^{\mathsf{m}} & := & \mathtt{TypeVal}\ (\mathcal{E}\ [\![A]\!]^{\mathsf{p}})\ (\mathtt{err}\ [\![A]\!]^{\mathsf{p}}) & [\{A\}^{\mathsf{p}}_{\mathsf{m}}]_{\varepsilon} & := & \lambda x : \mathcal{E}\ [\![A]\!]^{\mathsf{p}}.\ \mathtt{IsV}\ [\![A]\!]^{\mathsf{p}}\ x \\
[\{A\}^{\mathsf{e}}_{\mathsf{m}}]^{\mathsf{m}} & := & [A]^{\mathsf{e}} & [\{A\}^{\mathsf{e}}_{\mathsf{m}}]_{\varepsilon} & := & \lambda x : \mathtt{El}\ [A]^{\mathsf{e}}.\ \top
\end{array}
$$

Fig. 9. Translation of the four modalities

$$
\begin{array}{lll|lll}
[\iota^{\mathsf{m}}_{\mathsf{e}}]^{\mathsf{e}} & := & \lambda(A : \mathtt{type})\ (x : \mathtt{El}\ A).\ x & & & \\
[\iota^{\mathsf{m}}_{\mathsf{p}}]^{\mathsf{p}} & := & \lambda(A : \mathtt{type})\ (x : \mathtt{El}\ A).\ x & & & \\
[\iota^{\mathsf{p}}_{\mathsf{m}}]^{\mathsf{m}} & := & \lambda(A : \square)\ (x : A).\ \mathtt{val}\ A\ x & [\iota^{\mathsf{p}}_{\mathsf{m}}]_{\varepsilon} & := & \lambda(A : \square)\ (a : A).\ \mathtt{isV}\ A\ a \\
[\iota^{\mathsf{e}}_{\mathsf{m}}]^{\mathsf{m}} & := & \lambda(A : \mathtt{type})\ (x : \mathtt{El}\ A).\ x & [\iota^{\mathsf{e}}_{\mathsf{m}}]_{\varepsilon} & := & \lambda(A : \mathtt{type})\ (\_ : \mathtt{El}\ A).\ \mathtt{tt}
\end{array}
$$

Fig. 10. Translation of the four introduction operators

point of view, the problem is now that there is no way to give a RETT type to the failure premise, as the term $\mathtt{raise}\ \mathbb{B}^{\mathsf{m}}\ M$ is ill-typed. As a consequence, there is no `catch` term from $\square^{\mathsf{m}}$ into $\square^{\mathsf{p}}$.

4.5 Modalities

The translation of modalities is given in Figure 9. Notice that the composed modality that goes from $\square^{\mathsf{e}}$ to $\square^{\mathsf{p}}$ and back is not at all the identity, as it will add freely exceptions to a type that is already exceptional. This justifies the claim in Section 3 that no reasonable interplay is possible between the pure layer and the exceptional one. Indeeed, adding exceptions to CIC is a whole program translation that deeply modifies the structure of the program so that one cannot *internally* go back and forth between the two layers while preserving the program structure.

The term translation of $\{-\}^{\mathsf{m}}_{\mathsf{e}}$ is the identity, as it only consists in forgetting the parametricity witnesses when going from $\square^{\mathsf{m}}$ to $\square^{\mathsf{e}}$. The translation of $\{-\}^{\mathsf{m}}_{\mathsf{p}}$ is given by `El` as it consists in recovering the underlying type of an inhabitant of `type`.[3] The translation of $\{-\}^{\mathsf{p}}_{\mathsf{m}}$ is given by freely adding the exception type $\mathbb{E}$ to the base pure type using a sum type. Its parametricity predicate corresponds to witnesses that the inhabitant of the sum type is actually a value rather than an exception. To this end, we use the following dedicated inductive types in the target theory.

$$\mathtt{Inductive}\ \mathcal{E}\ (A : \square)\ : \square := \mathtt{val} : A \to \mathcal{E}\ A \mid \mathtt{err} : \mathbb{E} \to \mathcal{E}\ A$$

$$\mathtt{Inductive}\ \mathtt{IsV}\ (A : \square)\ : \mathcal{E}\ A \to \square := \mathtt{isV} : \Pi a : A.\ \mathtt{IsV}\ A\ (\mathtt{val}\ A\ a)$$

Finally, the translation of $\{-\}^{\mathsf{e}}_{\mathsf{m}}$ is given by the identity on the $[-]^{\mathsf{m}}$ part; its parametricity predicate is trivial, as described by the following inductive type in the target theory.

$$\mathtt{Inductive}\ \top\ : \square := \mathtt{tt} : \top$$

The translation of the introduction operators, presented in Figure 10, is straightforward, being either the identity or a canonical injection. Note in particular that $[\iota^{\mathsf{m}}_{\mathsf{e}}\ A\ (\mathtt{raise}\ A\ e)]_{\varepsilon} \equiv \mathtt{tt}$: exceptions can live in the mediating layer through the modality, as trivially harmless terms.

[3] We would like to define the translation as the combination of an element of the underlying type plus a proof that it is parametric, but we do not have access to the parametricity predicate in the $[-]$ translation. This definition will be made possible in Section 5 by considering a subtheory of RETT.

$$
\begin{array}{lcl}
[elim^{\mathtt{m}}_{\mathtt{e}}]^{\mathtt{e}} & := & \lambda(A : \mathtt{type})\,(P : \mathtt{El}\ A \to \mathtt{type})\,(P_a : \Pi a : \mathtt{El}\ A.\, \mathtt{El}\ (P\ a))\,(x : \mathtt{El}\ A).\, P_a\ x \\
[elim^{\mathtt{m}}_{\mathtt{p}}]^{\mathtt{p}} & := & \lambda(A : \mathtt{type})\,(P : \mathtt{El}\ A \to \square)\,(P_a : \Pi a : \mathtt{El}\ A.\, P\ a)\,(x : \mathtt{El}\ A).\, P_a\ x \\
[elim^{\mathtt{p}}_{\mathtt{m}}]^{\mathtt{m}} & := & \lambda(A : \square)\,(P : \mathcal{E}\ A \to \mathtt{type})\,(P_a : \Pi a : A.\, \mathtt{El}\ (P\ (\mathtt{val}\ A\ a)))\,(x : \mathcal{E}\ A). \\
 & & \mathtt{rec}_{\mathcal{E}}\ P\ (\lambda a.\, P_a\ a)\ (\lambda e.\, \mathtt{Err}\ (P\ (\mathtt{err}\ e)))\ x \\
[elim^{\mathtt{e}}_{\mathtt{m}}]^{\mathtt{m}} & := & \lambda(A : \mathtt{type})\,(P : \mathtt{El}\ A \to \mathtt{type})\,(P_a : \Pi a : \mathtt{El}\ A.\, \mathtt{El}\ (P\ a))\,(x : \mathtt{El}\ A).\, P_a\ x \\
[elim^{\mathtt{p}}_{\mathtt{m}}]_{\varepsilon} & := & \lambda(A : \square)\,(P : \mathcal{E}\ A \to \mathtt{type})\,(P_\varepsilon : \Pi(x : \mathcal{E}\ A)\, x_\varepsilon.\, \mathtt{El}\ (P\ x) \to \square). \\
 & & \lambda(P_a : \Pi a : A.\, \mathtt{El}\ (P\ (\mathtt{val}\ a)))\,(P_{a\varepsilon} : \Pi a : A.\, P_\varepsilon\ (\mathtt{val}\ A\ a)\ (\mathtt{isV}\ A\ a)\ (P_a\ a)). \\
 & & \lambda(x : \mathcal{E}\ A)\,(x_\varepsilon : \mathtt{IsV}\ A\ x). \\
 & & \mathtt{rec}_{\mathtt{IsV}}\ A\ (\lambda(x : \mathcal{E}\ A)\, x_\varepsilon.\, P_\varepsilon\ x\ x_\varepsilon\ ([elim^{\mathtt{p}}_{\mathtt{m}}]^{\mathtt{m}}\ A\ P\ P_a\ x))\ P_{a\varepsilon}\ x\ x_\varepsilon \\
[elim^{\mathtt{e}}_{\mathtt{m}}]_{\varepsilon} & := & \lambda(A : \mathtt{type})\,(P : \mathtt{El}\ A \to \mathtt{type})\,(P_\varepsilon : \Pi(a : \mathtt{El}\ A)\, a_\varepsilon.\, \mathtt{El}\ (P\ a) \to \square). \\
 & & \lambda(P_a : \Pi a : \mathtt{El}\ A.\, \mathtt{El}\ (P\ a))\,(P_{a\varepsilon} : \Pi a : \mathtt{El}\ A.\, P_\varepsilon\ a\ \mathtt{tt}\ (P_a\ a))\,(x : \mathtt{El}\ A)\,(x_\varepsilon : \top). \\
 & & \mathtt{rec}_{\top}\ (\lambda p.\, P_\varepsilon\ x\ p\ (P_a\ x))\ (P_{a\varepsilon}\ x)\ x_\varepsilon
\end{array}
$$

Fig. 11. Translation of the four eliminators of modalities

The translation of eliminators (Fig. 11) is more complex. The translations of $elim^{\mathtt{m}}_{\mathtt{e}}$ and $elim^{\mathtt{m}}_{\mathtt{p}}$ are almost the identity. The translations of $elim^{\mathtt{e}}_{\mathtt{m}}$ and $elim^{\mathtt{e}}_{\mathtt{m}}$ require the use of the eliminators to the inductive types introduced by the translation. $[elim^{\mathtt{p}}_{\mathtt{m}}]$ pattern-matches on the inhabitant of x of type $\mathcal{E}\ [A]$: if it is a value (i.e. of the form $\mathtt{val}\ A\ a$), it uses P_a given in the hypothesis; if it is an exception (i.e. of the form $\mathtt{err}\ A\ e$) it re-raises the exception of the return predicate. $[elim^{\mathtt{e}}_{\mathtt{m}}]$ is almost the identity. $[elim^{\mathtt{p}}_{\mathtt{m}}]_{\varepsilon}$ pattern-matches on the parametricity proof, which ensures that a parametric inhabitant of $\mathcal{E}\ [A]$ is equal to a term $\mathtt{val}\ A\ a$ for some a in A. The translation of $[elim^{\mathtt{e}}_{\mathtt{m}}]_{\varepsilon}$ is given by the fact that any x_ε in $\top$ is equal to $\mathtt{tt}$.

Note that the translation of modalities allows us to show a variant of Lemma 3.4, that the translation of an exceptional inductive type and its corresponding lowering are convertible.

Lemma 4.1. $[\{\mathbb{B}^{\mathtt{m}}\}^{\mathtt{m}}_{\mathtt{e}}]^{\mathtt{e}} \equiv [\mathbb{B}^{\mathtt{e}}]^{\mathtt{e}}$.

4.6 Parametricity Predicate

As explained in Section 3.3, any type in the exceptional layer of the form $\{A\}^{\mathtt{m}}_{\mathtt{e}}$ can be equipped with a parametricity predicate $\mathcal{P}\ A$ coming from $[A]_\varepsilon$. The translation of $[\mathcal{P}\ A]$ is just given by $\top$ as there is no information to provide at this stage, the parametricity predicate being available only for the parametric translation. The translation of $[\mathcal{P}\ A]_\varepsilon$ is simply returning the parametricity predicate $[A]_\varepsilon$ given by the translation. The translation of the elimination principle of $\mathcal{P}$ is straightforward. The parametricity predicate of types lifted from the exceptional layer is trivial, as both $[\mathcal{P}\ \{A\}^{\mathtt{m}}_{\mathtt{e}}] \equiv \top$ and $[\mathcal{P}\ \{A\}^{\mathtt{m}}_{\mathtt{e}}]_\varepsilon \equiv \top$, which explains why there is no way to extract any useful content from it.

Finally, any inductive type in the mediation layer (e.g. $\mathbb{B}^{\mathtt{m}}$) gives rise to a catch recursor on its lowering (e.g. $\{\mathbb{B}^{\mathtt{m}}\}^{\mathtt{m}}_{\mathtt{e}}$). Due to the fact that there is no difference in the underlying translation between $\mathbb{B}^{\mathtt{m}}$ and $\mathbb{B}^{\mathtt{e}}$, the translation of this recursor is the same as in Section 4.4, that is, it is given by pattern-matching with the additional failure case being handled by the additional failure premise.

4.7 Metatheoretical Properties of RETT

The soundness of the translations $[-]^s$ follow from the following properties.

Theorem 4.2 (Soundness). *The following properties hold.*

- $[M\{x := N\}]^s \equiv [M]^s\{x := [N]^s\}$ *(substitution lemma).*

$$
\begin{array}{lcl}
[\mathcal{P}]^{\mathsf{m}} & := & \lambda(A : \mathtt{type})\,(x : \mathtt{El}\ A).\,\top \\
[\iota_{\mathcal{P}}]^{\mathsf{e}} & := & \lambda(A : \mathtt{type})\,(x : \mathtt{El}\ A).\,\mathtt{tt} \\
[\Downarrow_{\mathcal{P}}]^{\mathsf{m}} & := & \lambda(A : \mathtt{type})\,(x : \mathtt{El}\ A)\,(p : \top).\,x \\
[\mathcal{P}]_{\varepsilon} & := & \lambda(A : \mathtt{type})\,(A_{\varepsilon} : \mathtt{El}\ A \to \square)\,(x : \mathtt{El}\ A).\,\top \to A_{\varepsilon}\ x \\
[\Downarrow_{\mathcal{P}}]_{\varepsilon} & := & \lambda(A : \mathtt{type})\,(A_{\varepsilon} : \mathtt{El}\ A \to \square)\,(x : \mathtt{El}\ A)\,(p : \top)\,(p_{\varepsilon} : A_{\varepsilon}\ x).\,p_{\varepsilon}
\end{array}
$$

Fig. 12. Translation of the $\mathcal{P}$ predicate

- *If* $M \equiv N$ *then* $[M]^s \equiv [N]^s$ *(conversion lemma).*
- *If* $\Gamma \vdash M : A$ *then* $[\![\Gamma]\!] \vdash [M]^s : [\![A]\!]^s$ *(typing soundness).*
- *If* $\Gamma \vdash A : \square^s$ *then* $[\![\Gamma]\!] \vdash [A]_{\varnothing} : \mathbb{E} \to [\![A]\!]^s$*, when* $s \in \{e, m\}$ *(exception soundness).*

Proof. The first property is by routine induction on M, the second is direct by induction on the conversion derivation. The third is by induction on the typing derivation. As in [Pédrot and Tabareau 2018], the most important rule is $\square_i^s : \square_j^s$, for the three layers. For the exception layer (and similarly the mediation layer), it holds because $[\square_i^{\mathsf{e}}]^{\mathsf{e}} \equiv \mathtt{TypeVal}\ \mathtt{type}_i\ \mathtt{TypeErr}_i$ has type $\mathtt{type}_j$ which is convertible to $[\![\square_j^{\mathsf{e}}]\!]^{\mathsf{e}}$. For the pure layer, it holds trivially because $[\square_i^{\mathsf{p}}]^{\mathsf{p}} \equiv \square_i$. For all the new constants in RETT that have not been considered in [Pédrot and Tabareau 2018], such as modalities and the parametricity predicate, one only has to check that their translations type check. The last property is a direct application of typing soundness. □

The parametric translation for terms and types that live in the mediation layer is also sound.

Theorem 4.3 (Parametricity Soundness). *The two following properties hold.*

- *If* $M \equiv N$ *then* $[M]_{\varepsilon} \equiv [N]_{\varepsilon}$.
- *If* $\Gamma \vdash M : A : \square_i^{\mathsf{m}}$ *then* $[\![\Gamma]\!]_{\varepsilon} \vdash [M]_{\varepsilon} : [\![A]\!]_{\varepsilon}\ [M]^{\mathsf{m}}$.

Proof. By induction on the derivation. The new typing rules in RETT that have not been considered in [Pédrot and Tabareau 2018] are in the definitions that crosscut the different layers. □

The fact that Theorems 4.2 and 4.3 hold on the whole translation of RETT into CIC allows us to automatically lift many metatheorical properties of CIC to RETT.

The first obvious one is the consistency of the mediation and pure layers.

Theorem 4.4 (Consistency). *The pure layer and the mediation layer of* RETT *are logically consistent.*

Proof. Theorem 4.2 on the pure layer guarantees that if M inhabits $\bot_{\mathsf{p}}$ in the pure layer, then $[M]^{\mathsf{p}}$ inhabits $[\bot_{\mathsf{p}}]^{\mathsf{p}} \equiv \bot$ in CIC. In the mediation layer, if M inhabits $\bot_{\mathsf{m}}$, then by Theorem 4.3, $[M]_{\varepsilon}$ inhabits $[\![\bot_{\mathsf{m}}]\!]_{\varepsilon}\ [M]^{\mathsf{m}} \equiv \bot_{\varepsilon}\ [M]^{\mathsf{m}}$, which is equivalent to $\bot$ because $\bot_{\varepsilon}$ has no constructor. □

The pure and mediation layers of RETT also enjoy a form of canonicity. Canonicity (for booleans) in CIC says that any closed term of type $\mathbb{B}$ is convertible to either `true` or `false`. In RETT, we do not know if canonocity holds for the standard conversion (that is, the equational theory arising from the usual rules of CIC together with the additional rules provided for RETT combinators), because this result amounts to showing the completeness of computational laws with respect to the new constants introduced. However, we can prove canonicity for a stronger form of conversion, namely the conversion induced by the translation in CIC, which is complete by definition:

$$\mathrm{M} \equiv_{[]} \mathrm{N} := [M] \equiv [N].$$

Theorem 4.5 (Canonicity). *The pure layer and the mediation layer of* RETT *enjoy canonicity for* $\equiv_{[]}$.

Proof. By Theorem 4.2, any closed term M of type $\mathbb{B}$ in the pure layer gives rise to a closed term $[M]^{\mathsf{p}}$ of type $\mathbb{B}$ in CIC. By canonicity, $[M]^{\mathsf{p}}$ is either convertible to `true` or `false`, but then as $[\mathtt{true}]^{\mathsf{p}} \equiv \mathtt{true}$ (and similarly for `false`), we have that $M \equiv_{[]} \mathtt{true}$ or $M \equiv_{[]} \mathtt{false}$ in RETT.

For the mediation layer, the situation is slightly more complicated. Theorem 4.3 guarantees that any closed term M of type $\mathbb{B}$ in the mediation layer gives rise to a closed term $[M]_\varepsilon$ of type $\mathbb{B}_\varepsilon\ [M]^{\mathsf{m}}$ in CIC. By canonicity of $\mathbb{B}_\varepsilon$ in CIC, this means that $[M]_\varepsilon$ is convertible to either $\mathtt{true}_\varepsilon$ or $\mathtt{false}_\varepsilon$, and so $[M]^{\mathsf{m}}$ is convertible to either $\mathtt{true}^\bullet$ or $\mathtt{false}^\bullet$. The property follows from the fact that $[\mathtt{true}]^{\mathsf{m}} \equiv \mathtt{true}^\bullet$ (and similarly for `false`). □

Beyond CIC. While we have formulated RETT as an extension of CIC, it is also interesting to consider extensions of CIC with certain axioms, such as function extensionality.

Interestingly, the translation of RETT satisfies a conservativity result for the pure layer, which states that every axiom that is compatible with CIC is also compatible with the pure layer. To state this theorem, we need to consider the trivial embedding $[-]_{\mathrm{CIC}}$ of CIC into RETT, which is defined by congruence everywhere but for the universes, with $[\square_i]_{\mathrm{CIC}} := \square_i^{\mathsf{p}}$.

Theorem 4.6 (Conservativity of the Pure Layer). *Given an axiom Ax, if* CIC $+$ *Ax is consistent, then the pure layer of* RETT $+$ $[Ax]_{\mathrm{CIC}}$ *is also consistent.*

Finally, we observe that this conservativity result does *not* hold for the mediation layer. For instance, function extensionality can be negated in the mediation layer. This has already been observed in Pédrot and Tabareau [2018].

Theorem 4.7 (Negation of function extensionality Pédrot and Tabareau [2018]). *Function extensionality is not valid in* $\square^{\mathsf{m}}$.

Proof. The two functions $\lambda x : \top.\, x$ and $\lambda x : \top.\, \mathtt{tt}$ can be distinguished in the mediation layer, because we can construct a parametric predicate that observes that the former re-raises exceptions, while the latter is constant and does not. □

In particular, the previous theorem shows that the mediation layer does not preserve univalence [Univalent Foundations Project 2013], which (coarsely) states that two equivalent types are equal. However, it can be shown using the translation that the mediation layer preserves Uniqueness of Identity Proof (UIP). This means that the mediation layer has to be considered with care when seen as a logical layer.

5 IMPLEMENTATION IN COQ

The translation of RETT into CIC can be seen as a compilation phase that extends the theory of Coq using a plugin, similarly to other syntactic translations [Jaber et al. 2016; Pédrot and Tabareau 2017, 2018]. By construction, this does not require any modification to the Coq kernel. Only two additional properties need to be trusted when using the plugin:

- First, that the soundness theorems 4.2 and 4.3 hold.
- Second, that the plugin implements the two translations correctly.

Due to the intrinsic nature of syntactic models, even this additional trust is relative. Soundness failure woud merely result in a Coq type error, as the translated terms are still checked by the (unmodified) kernel. Bogus mistranslation is more worrisome, because the compiled term could be unrelated to what the user had in mind. Thankfully, this kind of issue is very similar in spirit to Pollack-inconsistency [Wiedijk 2012], and can be worked around in the same way. Namely, the user can always pierce through the plugin abstraction layer, and scrutinize the translated proof term to check that it indeed corresponds to what was expected.

The specific problem to be solved for RETT is that it has three different hierarchies of universes, which is not the case in Coq. However, we can define a subtheory of RETT that only mentions $\square_i^{\mathsf{e}}$ and $\square_i^{\mathsf{p}}$. By further assuming that $\square^{\mathsf{p}}$ is impredicative, we can implement a plugin that adds exceptions to Coq, where the universe hierarchy `Type` is interpreted as the hierarchy of exceptional types $\square^{\mathsf{e}}$, and the impredicative universe `Prop` is interpreted as the (impredicative) universe of pure types $\square^{\mathsf{p}}$.[4] The mediation layer $\square^{\mathsf{m}}$ is omitted, but its internal parametricity predicate $\mathcal{P}$ is realized as a Coq type class [Sozeau and Oury 2008].

In this section, we first present how to implement the plugin in Coq. We then explain how we use the type class mechanism to represent parametricity. Finally, we come back to the examples introduced in Section 1.

The code of the example of this section can be found in the file `list_theorem.v` of the anonymous supplementary material.

5.1 CoqRETT: RETT as a Coq plugin

We define a Coq plugin that implements the translation described in Section 4 and provides new constructors in Coq, giving them meaning through the translation. Currently, the plugin does not instrument all the constants of CoqRETT so the user needs to define those constants explicitly.

To define a new constant `C:A` in CoqRETT, we need to provide a constant of the translation of `[C]:[A]` in Coq. This is done using the command

```
Effect Definition C : A.
(** definition of [C] **)
Defined.
```

When working inside CoqRETT as a source theory, new definitions can then be introduced as in standard Coq.

The basic new primitives that are available in CoqRETT are the type of exceptions and the function that raises an exception at any type.

```
Definition Exception : Type.
Definition raise : ∀ A : Type, Exception → A.
```

Note that because of the cumulativity of universes in Coq, there is no way to prevent a user to raise an exception in `Prop`, as `Prop` is a subtype of `Type`. However, translating such an exception will produce a translation error. This means that the correctness of a proof in CoqRETT is not guaranteed only by typechecking the proof, but by additionally typechecking the translation of the proof.[5]

When we define an inductive type in `Type`, for instance lists

```
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A → list A → list A
```

we generate the standard eliminator on `Type`, but we can also provide the catch eliminator on `Type` giving it meaning with the translation:

[4] Note that impredicativity is orthogonal to purity; we just exploit the existence of two separate universe hierarchies in Coq.
[5] We could make this more transparent to the user by instrumenting typechecking to perform both standard typechecking of the term and of its translation.

```
Effect Definition list_catch : ∀ A (P : list A → Type),
P nil → (∀ (a : A) (l : list A), P l → P (a :: l)) → (∀ e, P (raise A e)) → ∀ l : list A, P l.
```

Similarly, we can define the corresponding eliminator `list_catch_prop` in `Prop`.

After going through the translation phase, the computation laws of `list_catch` can be proven by reflexivity, which means that they are indeed definitionally valid in RETT. For instance,

```
Effect Definition list_catch_nil_eq : ∀ A (P : list A → Type) Pnil Pcons Praise,
 list_catch A P Pnil Pcons Praise nil = Pnil.
Proof.
 reflexivity.
Defined.
```

However, these laws can only be proven for *propositional equality* in CoqRETT. This is a limitation in the usability of the plugin, due to the fact that a Coq plugin cannot extend the conversion of the Coq kernel. Indeed, RETT extensions are defined as axioms in the CoqRETT surface language, and computing with them would require a way to make these equalities *definitional*.[6] Therefore, explicit rewriting with these equalities is necessary when staying in the source theory.

Using the catch eliminator, it is already possible to prove internally in CoqRETT, for instance, that the empty list `nil A` can be discriminated from `raise (list A) e`.

```
Definition nil_not_raise: ∀ A e, nil A ≠ raise e.
Proof.
 intros A e.
 assert (H:∀ l', nil A = l' → list_catch _ _ True (fun _ _ _ ⇒ False) (fun _ ⇒ False) l').
 { intros l' eq. destruct eq. rewrite list_catch_nil_eq. exact I. }
 intro eq. specialize (H (raise e) eq). rewrite list_catch_raise_eq in H. exact H.
Defined.
```

As usual in Coq, the proof relies on dependent elimination. It starts by generalizing `nil A ≠ raise e` to `∀ l', nil A = l' → list_catch _ _ True (fun _ _ _ ⇒ False) (fun _ ⇒ False) l'`. Of course, when `l'` is `raise e`, this is the same proposition, but generalizing it allows us to do elimination on the proof of equality, which tells us that we must be in the `nil A` case. Note that in the proof, we need to do explicit rewriting with `list_catch_nil_eq` because `list_catch` does not compute. Once the generalization is proven, the property follows by specialization and rewriting.

5.2 $\mathcal{P}$ as a Type Class

The internal parametricity predicate $\mathcal{P}$ of the mediation layer is realized in CoqRETT as a type class. Indeed, because not every type in $\square^{\mathsf{e}}$ is of the form $\{A\}^{\mathsf{m}}_{\mathsf{e}}$ for some A in $\square^{\mathsf{m}}$, in general, the parametricity predicate is not defined on every type.

The `Param` type class is used to denote parametric terms.

```
Class Param (A : Type) : Type := { param : A → Prop }
```

The plugin automatically generates instances of the `Param` type class for inductive types using the parametric translation. Note that we do not make any distinction between the type of lists that comes from the mediation layer and the type of lists in the exceptional layer. This is valid because the two types are isomorphic (Lemma 3.4) and even translated to the same type (Lemma 4.1).

[6] Extending the reach of the plugin architecture of Coq to support new conversion rules is an interesting perspective, although far beyond the scope of this work.

In order to be able to exploit the parametricity predicate for reasoning on inductive types, we add a class ParamInd, which provides the param_correct property for instances of Param on inductive types.

```
Class ParamInd (A : Type) `{Param A} : Type :=
  { param_correct : ∀ e : Exception, param (raise A e) → False }
```

This property captures the reasoning principle that when a term is parametric, it cannot be an exception. This corresponds to Lemma 3.5 in RETT, here expressed as a primtive notion.

We can recover the eliminator restricted to parametric terms defined in Section 3.5 by using induction on the catch eliminator and the param_correct property. In the case of recursive inductive types, we first need to provide an inversion principle that the parametricity of a term implies the parametricity of its subterms, which in the case of lists amounts to

```
Effect Definition param_list_cons : ∀ A a (l:list A), param (cons a l) → param l.
```

Then, the eliminator restricted to parametric terms can be defined as

```
Definition list_ind : ∀ A (P : list A → Prop),
    P nil → (∀ (a : A) (l : list A), P l → P (a :: l)) → ∀ l : list A, param l → P l.
Proof.
 intros A P Pnil Pcons l; induction l using list_catch_prop.
 - intro. exact Pnil.
 - intros param_al. exact (Pcons a l (IHl (param_list_cons _ _ _ param_al))).
 - intros param_e. destruct (param_correct e param_e).
Defined.
```

5.3 Back to Examples

Let us come back to the examples of Section 1, explaining how to state and prove the described results. We insist that all the reasoning that follows is done directly in CoqRETT (as opposed to in Coq over the results of the translation). In the examples we fix the exception type to strings.

Tail of Non-Empty Lists. We can prove in CoqRETT that the exception-raising tail function

```
Definition tail {A} (l : list A) : list A :=
  list_rect (fun _ ⇒ list A) (raise "error: empty list") (fun _ l _ ⇒ l) l.
```

does not raise an exception when applied to a non-empty list.

To prove the tail property, we first need to establish that, when an integer is provably bigger than another integer, it cannot be an exception. This is because there is no constructor for exceptions in the definition of ≤—comparison is an inductively-defined predicate in Prop, and is therefore pure.

```
Definition raise_not_leq : ∀ (n:ℕ) e, n ≤ raise e → False.
```

This discrimination property (directly induced by the catch eliminator) allows us to prove the non-failing behavior of tail on non-empty lists.

```
Definition non_empty_list_distinct_tail_error: ∀ A e (l: list A),
   length l > 0 → tail l ≠ raise e.
Proof.
 intros A e l; induction l using list_catch_prop; cbn.
 - inversion 1.
 - intros Hlen eq. apply le_S_n in Hlen. eapply raise_not_leq. rewrite eq in Hlen.
   rewrite list_rect_raise_eq in Hlen. exact Hlen.
```

```
  – intros Hlen. unfold length in Hlen. rewrite list_rect_raise_eq in Hlen.
    destruct (raise_not_leq _ _ Hlen).
Defined.
```

The proof is quite direct using induction on the catch eliminator and the `raise_not_leq` discrimination property. The only additional reasoning burden is due to the absence of computation rules for catch elimination, as discussed previously.

Head of Non-Empty Lists. Let us now turn to the exception-raising head function

```
Definition head {A} (l: list A) : A :=
  list_rect (fun _ ⇒ A) (raise "error: empty list") (fun a _ _ ⇒ a) l.
```

Recall that proving that applying `head` on a non-empty list does not produce an exception requires a *deep* notion of parametricity for lists. Deep parametricity is necessary to say that not only the shape of the list is non-exceptional, but also that its contained values are non-exceptional. Of course, this notion of deep parametricity only makes sense for element types for which there exists an instance of the `Param` type class. The predicate `list_param_deep` below defines deep parametricity for lists:

```
Definition list_param_deep A {H: Param A} : ∀ (l: list A), Prop :=
 list_catch A (fun _ : list A ⇒ Prop)
          True
          (fun (a : A) (_ : list A) (lind : Prop) ⇒ param a ∧ lind)
          (fun _ : Exception ⇒ False).
```

It uses the catch eliminator, returning `True` in the case of an empty list and `False` in the case of an exception. The difference with the shallow parametricity predicate is in the recursive case `cons a l`: we require both `a` and `l` to be parametric, the former using the instance of `Param` in hypothesis, and the latter with a recursive call to `list_param_deep`.

With this extra assumption on the list, we can now state and prove the correctness property of head for non-empty lists.

```
Definition head_empty_list_no_error: ∀ A {H: Param A} e (l: list A),
  length l > 0 → list_param_deep l → head l ≠ raise e.
Proof.
 intros A A_param e l. induction l using list_catch_prop.
 – inversion 1.
 – intros Hlen Hl. unfold list_param_deep in Hl.
   rewrite list_catch_cons_eq in Hl. cbn in *.
   destruct Hl as [Ha _]. intro eq. rewrite eq in Ha. apply (param_correct e Ha).
 – intros. unfold length in H. rewrite list_rect_raise_eq in H. compute in H.
   destruct (raise_not_leq _ _ H).
Defined.
```

The proof is again quite direct using induction on the catch eliminator and the `raise_not_leq` discrimination property. The only extra reasoning is in the case the list is actually of the form `cons a l`, where we use the deep parametricity of the list to know that `a` is actually parametric, which by `praram_correct` means that it cannot be an exception.

6 RELATED WORK

This work relates to the large body of work on integrating effects and dependent types. Hoare Type Theory (HTT) [Nanevski et al. 2008], used in particular in the Ynot project [Chlipala et al. 2009], is realized as an axiomatic extension of Coq with effects encapsulated in a Hoare monad. HTT does not address the main challenge of effectful terms at the type level because it essentially only supports proving in Coq properties on *simply-typed* imperative programs. Dependent ML [Xi and Pfenning 1999] also side-steps the issue by only allowing types to depend on pure terms, namely arithmetic expressions that denote array lengths. The $F^\star$ programming language [Swamy et al. 2016] uses a notion of primitive effects including state, exceptions, divergence and IO. Each effect is described through a monadic predicate transformer semantics. The use of monads makes it possible to isolate a pure core dependent language to reason about effectful programs. However, the standard monadic approach [Moggi 1991] does not scale to dependent types, because one cannot provide a dependently-typed version of the bind operation. Idris [Brady 2013] favors algebraic effects instead of monads as an elegant way to combine effects and dependent types, though with the same restrictions. In contrast, RETT supports reasoning about exceptional programs that can make use of the full power of dependent types.

An alternative, and much lower-level way to address the issue is to represent the effectful fragment of the type theory as a deep embedding of the syntax of this fragment inside the theory. This happens commonly in the implementation of compilers in some flavor of type theory, like e.g. CompCert [Leroy et al. 2016] or Cake ML [Kumar et al. 2014]. While this approach is extremely simple and readily available in weak theories such as LF, it is completely oblivious of the advantages of dependent types. That is, the equational rules of the embedded language have to be computed explicitly in the proof, which in turn also requires proving that the properties of the host language are stable under these rules. As such, handling advanced features like higher-order functions is painful, let alone the preservation of typing of the various programs being considered.

RETT builds upon the translation approach to extend type theory non-axiomatically [Boulier et al. 2017]. Internal translations of type theory have a fairly extensive history. Barthe et al. [1999] describe a CPS translation for CC_ω extended with `call/cc`, which does not handle inductive types. A variant of this translation that supports dependent sums using answer-type polymorphism was developed by Bowman et al. [2018]. Jaber et al. [2016] use forcing to define a generic class of internal translations of type theory that only work on a restricted version of dependent elimination. This limitation also applies to the Baclofen type theory [Pédrot and Tabareau 2017]. RETT is an extension of the Exceptional Type Theory (ETT) [Pédrot and Tabareau 2018], which was the first *complete* internal translation of CIC that adds a specific effect. As discussed earlier, ETT does not support consistent reasoning about exceptional terms; RETT addresses this limitation through a layered universe architecture with modalities. Both ETT and RETT rely on the internal translation presentation of parametricity of Bernardy and Lasson [2011] in order to impose observational purity on exceptional terms.

A promising venue to reconcile dependent types and effects is to study dependent variants of call-by-push-value (CBPV) [Levy 2001], as recently done by Ahman et al. [2016] and Vákár [2015]. While the CBPV setting can accommodate any effect described in monadic style, these approaches also need to impose a purity restriction for dependency. In contrast, the separation of the pure, mediation, and exceptional layers in RETT makes it possible to isolate restrictions to specific layers, allowing for instance the exceptional layer to freely mix effects and dependencies.

Finally, the Zombie language [Casinghino et al. 2014] combines proofs and potentially-diverging programs by clearly separating two fragments of the language. This separation is not unlike the layers of RETT, although Zombie does not make it possible to consistently reason about effectful terms. RETT provides solid type-theoretic foundations that can inform the design of similar practical dependently-typed programming languages, as illustrated by the design of the CoqRETT instantiation. In particular, the need for a mediation layer from which the parametricity predicate can be obtained is a key novelty of this work.

7 CONCLUSION AND FUTURE WORK

The Reasonably Exceptional Type Theory (RETT) supports consistent reasoning about exceptional programs in a full dependently-typed setting. As such, it promises to alleviate the task of developing and proving properties about programs that are inherently partial, as well as easing the interoperability between pure type theories used in proof assistants, and mainstream impure functional languages like OCaml and Haskell.

A key element of RETT is its integration of three universe hierarchies, clearly separating the pure and exceptional types, and introducing a mediation layer in order to allow both to interact in a sound manner. We believe this general approach could be beneficial in order to integrate other effects into type theories, sacrificing neither consistency nor modularity.

If this turns out to work for more general effects, it would also mean that it would be possible to extend type theory with *à la carte* effect systems. More precisely, for every single effect being considered, there would be a corresponding universe hierarchy, together with elimination principles that would provide ways to communicate between those different worlds. Such a presentation would be compatible with the current implementation of proof assistants such as Coq, and it would be easy to cherry-pick the particular subsystem one would like to work in.

Finally, the instantiation and implementation of RETT in Coq reveals the interest of a more powerful extension mechanism that would allow some selected propositional equalities to be treated definitionally. Currently, the plugin forces one to rely on explicit rewriting when using hand-defined RETT primitives, which is a major practical hurdle. Recent work on so-called *rewrite rules* [Cockx and Abel 2016] suggests that the full RETT theory can be emulated, definitional equations included, with a relatively self-contained extension of the Coq kernel. With such an extension, the plugin would be turned into a tiny shell generating the axioms induced by the translation with their associated rewrite rules. An alternative, but much more invasive solution would be to implement RETT directly in the Coq kernel. While not outright impossible, this would represent a massive amount of work, conflicting with other kinds of extensions like univalence.

REFERENCES

Danel Ahman, Neil Ghani, and Gordon D. Plotkin. 2016. Dependent Types and Fibred Computational Effects. In *19th International Conference on Foundations of Software Science and Computation Structures*. Springer Berlin Heidelberg, Eindhoven, The Netherlands, 36–54. DOI: http://dx.doi.org/10.1007/978-3-662-49630-5_3

Gilles Barthe, John Hatcliff, and Morten Heine B. Sørensen. 1999. CPS Translations and Applications: The Cube and Beyond. *Higher Order Symbol. Comput.* 12, 2 (Sept. 1999), 125–170. DOI: http://dx.doi.org/10.1023/A:1010000206149

Jean-Philippe Bernardy and Marc Lasson. 2011. Realizability and Parametricity in Pure Type Systems. In *Foundations of Software Science and Computational Structures*, Vol. 6604. Saarbrücken, Germany, 108–122. DOI: http://dx.doi.org/10.1007/978-3-642-19805-2

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

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

William Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. 2018. Type-Preserving CPS Translation of Σ and Π Types is Not Not Possible. In *Proceedings of the 45st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '18)*. ACM, New York, NY, USA.

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 05 (2013), 552–593.

Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, and Stephanie Weirich. 2018. Ready, Set, Verify! Applying hs-to-coq to Real-World Haskell Code (Experience Report). *Proceedings of the ACM on Programming Languages* 2, ICFP (Sept. 2018), 89:1–89:16.

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

Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. 2009. Effective Interactive Proofs for Higher-order Imperative Programs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA, 79–90. DOI: http://dx.doi.org/10.1145/1596550.1596565

Jesper Cockx and Andreas Abel. 2016. Sprinkles of Extensionality for Your Vanilla Type Theory. In *22nd International Conference on Types for Proofs and Programs (TYPES 2016)*.

Thierry Coquand and Gérard P. Huet. 1988. The Calculus of Constructions. *Inf. Comput.* 76, 2/3 (1988), 95–120. DOI: http://dx.doi.org/10.1016/0890-5401(88)90005-3

Guilhem Jaber, Gabriel Lewertowski, Pierre-Marie Pédrot, Matthieu Sozeau, and Nicolas Tabareau. 2016. The Definitional Side of the Forcing. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*. 367–376. DOI: http://dx.doi.org/10.1145/2933575.2935320

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Principles of Programming Languages (POPL)*. ACM Press, 179–191. DOI: http://dx.doi.org/10.1145/2535838.2535841

Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert – A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems*. SEE.

Pierre Letouzey. 2004. *Programmation fonctionnelle certifiée : l'extraction de programmes dans l'assistant Coq*. Ph.D. Dissertation. Université Paris XI.

Paul Blain Levy. 2001. *Call-by-push-value*. Ph.D. Dissertation. Queen Mary, University of London.

Assia Mahboubi and Enrico Tassi. 2008. *Mathematical Components*.

Eugenio Moggi. 1991. Notions of Computation and Monads. *Information and Computation* 93, 1 (July 1991), 55–92. DOI: http://dx.doi.org/10.1016/0890-5401(91)90052-4

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. 2008. Hoare type theory, polymorphism and separation. *Journal of Functional Programming* 18, 5-6 (2008), 865–911. DOI: http://dx.doi.org/10.1017/S0956796808006953

Ulf Norell. 2009. Dependently Typed Programming in Agda. In *Advanced Functional Programming (AFP 2008) (LNCS)*, Vol. 5832. Springer, 230–266.

Pierre-Marie Pédrot and Nicolas Tabareau. 2017. An effectful way to eliminate addiction to dependence. In *32nd Annual Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. 1–12. DOI: http://dx.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)*, Amal Ahmed (Ed.), Vol. 10801. Thessaloniki, Greece, 245–271.

The Coq Development Team. 2019. The Coq Proof Assistant Reference Manual. (2019). https://coq.inria.fr/refman

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism.. In *IFIP Congress* (2002-01-03). 513–523.

Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *Proceedings of the 21st International Conference on Theorem Proving in Higher-Order Logics*. Montreal, Canada, 278–293.

Nikhil Swamy, Cătălin Hriţcu, 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-Monadic Effects in F*. In *43nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 256–270. https://www.fstar-lang.org/papers/mumon/

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

Univalent Foundations Project. 2013. *Homotopy Type Theory: Univalent Foundations for Mathematics*. http://homotopytypetheory.org/book.

Matthijs Vákár. 2015. A Framework for Dependent Types and Effects. (2015). arXiv:arXiv:1512.08009 draft.

Freek Wiedijk. 2012. Pollack-inconsistency. *Electronic Notes in Theoretical Computer Science* 285 (2012), 85 – 100. DOI: http://dx.doi.org/10.1016/j.entcs.2012.06.008 Proceedings of the 9th International Workshop On User Interfaces for Theorem Provers (UITP10).

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