



UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

A REASONABLY EXCEPTIONAL TYPE THEORY:
A TYPE THEORY WITH EXCEPTIONS

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN CIENCIAS,
MENCIÓN COMPUTACIÓN

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN
COMPUTACIÓN

HANS JACOB FEHRMANN ROJAS

PROFESOR GUÍA:
ÉRIC TANTER

MIEMBROS DE LA COMISIÓN:
GONZALO NAVARRO BADINO
JOCELYN SIMMONDS WAGEMANN
PABLO BARCELÓ BAEZA

SANTIAGO DE CHILE
2020

Resumen

Los asistentes de pruebas son una herramienta de software donde es posible desarrollar programas sobre los cuales se pueden demostrar propiedades y así certificarlos. Esto es posible a través de teorías de tipo altamente expresivas que sirven como marcos de trabajo lógicos, donde pruebas y programas coexisten. Por otro lado, los lenguajes de programación modernos y sus sistemas de tipo asociados soportan, de alguna forma u otra, el manejo de excepciones. Sin embargo, los asistentes de pruebas no proveen un mecanismo de excepciones debido a la naturaleza de estos: las excepciones pueden producir sistemas inconsistentes (las pruebas no son válidas), lo que invalida el objetivo de un asistente de pruebas. De esta forma, no es posible utilizar excepciones en programas. Para el caso del *Calculus of Inductive Constructions* (CIC), la teoría de tipos usada por el asistente de pruebas Coq, existen trabajos previos que han añadido excepciones a la teoría. Sin embargo, estas extensiones resultan incómodas debido a que se pierden propiedades importantes de la teoría, como es la consistencia de la teoría, o no es posible razonar sobre excepciones de manera directa. El presente trabajo crea una nueva teoría de tipo, la *Reasonably Exceptional Type Theory* (RETT), donde es posible trabajar con excepciones y razonar sobre ellas de forma simple, directa y consistente. Esta teoría diferencia entre sus los elementos computacionales y sus los elementos proposicionales (o teoremas), donde los elementos computacionales siempre pueden ser excepciones mientras que los teoremas siempre son puros (no son excepciones). Adicionalmente, la teoría provee un mecanismo que permite expresar que un elemento computacional debe ser puro. La validación de esta teoría se realiza por una traducción sintáctica hacia CIC. Esta traducción provee buenas propiedades meta teóricas, lo que permite su uso como teoría subyacente para un asistente de pruebas con excepciones. Además, se desarrolló un plugin para Coq, el cuál hace disponible la teoría a los programadores. De esta forma, RETT es la primera teoría de tipos con excepciones consistente que permite el razonamiento sobre estas, con una implementación para Coq.

Abstract

Proof assistants are software tools where it is possible to develop certified programs: programs along proofs of the desired properties. This is made possible by highly expressive type theories that serve as logical frameworks, where proofs and programs coexist. On the other hand, modern programming languages and their type systems support, in one way or another, exception handling. However, proof assistants do not provide an exception mechanism due their nature: exceptions can produce inconsistent systems (proofs are not valid), which invalidates the objective of a proof assistant. Thus, it is not possible to use exceptions on programs. For the *Calculus of Inductive Constructions* (CIC), the core type theory of the proof assistant Coq, there are previous works that have added exceptions to it. However, these extensions are cumbersome because important properties of the theory are lost, such as the consistency of the theory, or it is not possible to reason about exceptions directly. The present work creates a new type theory, the *Reasonably Exceptional Type Theory* (RETT), where it is possible to work with exceptions and reason about them simply, directly and consistently. This theory differentiates between its computational elements and its propositional elements (or theorems), where computational elements can always be exceptions while theorems are always pure (not exceptions). Additionally, the theory provides a mechanism that allows expressing that a computational term must be pure. The validation of this theory is done by a syntactic translation into CIC. This translation provides good meta-theoretical properties, which allows its use as the core theory for a proof assistant with exceptions. Further, we developed a plugin for Coq, which makes the theory available for programmers. Thus, RETT is the first consistent type theory with exceptions that allows reasoning about them, with an implementation in Coq.

A Laura, el amor de mi vida, y a las gordas

Agradecimientos

En primer lugar, quiero agradecer a Laura, el amor de mi vida, por toda la paciencia y apoyo que me otorgó en este largo periodo.

Agradecer además el apoyo de mi familia y amigos, y todas las bromas (pesadas) sobre mi entrega del presente trabajo.

Finalmente, agradecer a Éric y su paciencia (siempre estuvo dispuesto a resolver cualquier inquietud, a pesar de que no fui el alumno más aplicado), a Nico y PIM por los *insights* en el desarrollo e implementación, y a Ren por dejar este documento lleno de comentarios rojos.

¡Gracias totales!

Contents

1	Introduction	1
2	Background	4
2.1	Lambda Calculus	4
2.2	Typed Lambda Calculus	8
2.2.1	Type System	8
2.2.2	Properties of the Simply-Typed Lambda Calculus	9
2.2.3	Beyond the Simply-Typed Lambda Calculus	10
2.2.4	Calculus of Constructions	11
2.3	Calculus of Inductive Constructions	11
2.3.1	Terms	11
2.3.2	Type System	12
2.3.3	Conversion Rules	14
2.3.4	Inductive Types	14
2.4	Coq: The Proof Assistant	21
2.5	Adding Exceptions to CIC	22
2.5.1	Models and Syntactic Translation	22
2.5.2	The Exceptional Type Theory	22
2.5.3	The Parametric Exceptional Type Theory	23
3	Reasonably Exceptional Type Theory: RETT	25
3.1	Motivation	25
3.2	System Definition	26
3.3	Reasonably Exceptional Translation Definition	26
3.3.1	Base Structure	27
3.3.2	Negative Fragment	28
3.3.3	Inductive Types	37
3.4	Examples	39
4	Controlling Exceptions: A Modality over RETT	41
4.1	Motivation	41
4.2	Reasonably Parametric Exceptional Translation	42
4.3	Parametric Modality	52
4.3.1	Extending the Target Theory	52
4.3.2	Extending the Source Theory: Adding the Modality	53
4.4	Modality Realization	55

5	RETT Implementation: A Coq Plugin	58
5.1	CoqRETT in Action	58
5.2	Limitations	62
5.3	Foundation Code	62
5.4	Plugin foundation	63
5.5	Plugin Implementation	64
5.6	Plugin Performance	66
6	Conclusion	68
	Bibliography	70

Chapter 1

Introduction

Certified programming is an emerging field, where the written programs are accompanied with a proof that they meet their specification. In other words, it is possible to establish properties over the programs. These properties can be seen as mathematical theorems that are machine-verified, all within the same programming language. This process can be carried out in *proof assistants* such as Coq [26], Agda [25], Isabelle [30]. The focus of our work is Coq.

Coq is a pure, functional, strictly typed programming language. Here, the theorems we are able to define are actually types. This fact allows the interplay between proofs and programs in the same programming language. For example, we can prove the associative property of addition for natural numbers. First, we need to define what a natural number is. Coq allows defining new types through its *inductive types* definition. The syntax to declare inductive types is similar to the syntax for new data types in OCaml or Haskell:

```
Inductive nat :=  
  | 0: nat  
  | S: nat → nat.
```

Here, we are saying that there are only two ways to construct a natural number `nat`: either being `0`, which represents zero, or a successor `S n` for some natural number `n`. We can then define the addition function as:

```
Fixpoint addition (n: nat) (m: nat): nat :=  
  match n with  
  | 0 ⇒ m  
  | S n0 ⇒ addition n0 (S m)  
end.
```

Here, we are saying that `addition` is a function that takes two parameters of type `nat` and returns a `nat` (the type after the colon). The body of the function does pattern matching on the first argument. Note that we must consider all constructors related to the type and must return the expected type for each branch, which is `nat` in this case. The pattern matching translates like this: if `n` is zero, then return the second parameter. If `n` is a successor of some natural number `n0`, then call recursively `addition` with parameters `n0` and the successor of `m`,

which is $S\ m$. We can establish that the expression $n + m$ is equal to `addition n m`. With these definitions, we can express the associative property as:

Theorem `addition_is_associative`: $\forall (n\ m: \text{nat}), n + m = m + n$.

The previous theorem is actually a function that receives two parameters `n` and `m` of type `nat` and produces a proof of associativity for them. The equality (`=`) is an inductive type that expects two parameters of the same type and returns a proposition. It represents that those two objects are *equal*. Thus, we are certifying a program.

An important feature of programming languages is exception handling. Exceptions are useful in programming languages because they make explicit unexpected results and can transfer execution in a non-local way. In languages like Java, OCaml and Python there is a primitive structure in the language that defines how exceptions must be handled in that scope: the **try/catch** block. Additionally, the language also provides a mechanism to raise exceptions: **throw**. The essence of the previous structures is the following:

```
try {
  ...
  if fileCouldNotBeOpen { throw IOException(); }
  ...
} catch (IOException) {
  ...
}
```

Haskell, on the other hand, takes a more pure approach: it does not provide a built-in mechanism to handle exceptions. Instead, Haskell leverages on its type system and its *do notation* to provide exception handling to the user: the monadic approach. In practice, the types **Maybe** or **Either** can be used as *exceptions*, which can be transformed, raised and handled:

```
errorFunction :: String -> Maybe Int
errorFunction fileName = do
  file <- openFile fileName
  content <- read file
  return (numberOfLines content)
```

It is natural to ask: what is the mechanism that Coq provides for exception handling? Let us assume the inductive type `list`:

```
Inductive list (A: Type) :=
| nil: list A
| cons: A -> list A -> list A.
```

where `nil` is the empty list and `cons a l` represents a new list with `a` on top of the other elements of `l`. We would like to define the `head` function of lists to throw an exception if the list is empty:

```

Definition head (A: Type) (l: list A): A :=
  match l with
  | nil => raise 'empty list'
  | cons a _ => a
end.

```

Sadly, Coq does not provide such a mechanism because it conflicts with its ability to certify programs. Coq is the implementation of dependent type theory: the *Calculus of Inductive Constructions* (CIC). In CIC, every function must be total and strictly typed. If CIC allows indiscriminate exceptions like the `head` case, it loses important properties that open the door to logical inconsistency. Coq supports the monadic approach to some extent, but it leads to a cumbersome usage. Also, the monadic approach does not scale to the whole system due to dependent types in CIC. Is it possible to add *exceptions* to Coq that can be used as in the `head` function?

If exceptions were available in Coq, they could relieve the imposed totality of the language: just raise an exception. Additionally, we could reason about programs that raise exceptions and establish properties about them. However, proofs are programs, so there could be exceptional proofs: proofs made valid by an exception. This means that exceptions must be handled with care at the propositional level because we want that proofs to remain valid while still being able to reason about them.

There are previous works that added exceptions to CIC. One of these works introduced the *Exceptional Type Theory* (ETT) [17]. ETT provides call-by-name exceptions to use freely in the language. Moreover, we can reason about exceptions. However, this theory is not logically consistent. In an attempt to recover logical consistency, [17] also introduced the *Parametric Exceptional Type Theory* (PETT). PETT recovers logical consistency, but it does not allow formulating theorems about exceptions.

This work presents a new type theory, the *Reasonably Exceptional Type Theory* (RETT) that provides exceptions, maintains logical consistency and allows reasoning about exceptions. Besides, RETT provides a mechanism to request that an exceptional element of the theory must not be an exceptions on-demand through a modality. Finally, we developed CoqRETT, an implementation of RETT, as a plugin for Coq.

We review the background required for understanding our work in Chapter 2. Second, we present the translation that generates RETT along its soundness in Chapter 3. We then develop a modality over RETT which enables ad-hoc purity of terms in Chapter 4. Finally, we realize the theory for Coq as a plugin in Chapter 5 along examples.

An extension of this work has been published at ICFP 2019 [19]. The present work is more tailored to the plugin implementation.

Chapter 2

Background

Our work is the development of a new type theory, but as stated before, type theories can also be seen as programming languages. We take the approach of presenting CIC and its foundations from a programming language perspective. Then, we present the base type theories of our work, ETT and PTT.

2.1 Lambda Calculus

Programming languages are complex. In order to capture these complexities, it is necessary to define abstractions. The lambda calculus is one.

In order to develop mathematics, it is necessary to define a context in which such definitions make sense. This context, however, can be implicit or be formulated in natural language. This motivated Church in the late 30's to create a logical framework in which every variable and proposition should state explicitly the context in which it is defined. The logical framework he created was the lambda Calculus (λ -calculus) [5].

Church intended to use this framework as a foundation of mathematics, but it was demonstrated logically inconsistent [13]. It then was used to capture the notion of computation [7] and demonstrated equivalent to the Turing machine.

The λ -calculus is composed of three kinds of expression.

- **Variables:** x, y, z, \dots
- **Applications:** $t_1 t_2$
- **Abstractions:** $\lambda x. t$

These expressions can be composed to form the *terms* of the calculus. We present them in Backus-Naur form:

$$t, u := x \mid \lambda x. t \mid t u$$

The term x represents a variable. It can also be thought as an identifier for some expression. The term $\lambda x. t$ represents a function (abstraction). It takes a parameter x and a body t , where

x can be used in t . Lastly, $t_1 t_2$ represents an application, i.e., apply the term t_2 to t_1 . The following expressions are valid terms:

$$x \quad y \quad \lambda x. x \quad \lambda x. \lambda y. x \quad \lambda z. z (\lambda x. z) \quad (\lambda z. z) (\lambda x. z) (\lambda y. y)$$

but these are not:

$$\lambda \quad \lambda x. \quad \lambda x. \lambda$$

Additionally, terms are left associative, which means that the terms

$$(\lambda z. z) ((\lambda x. z) x) y (\lambda x. x) \quad \lambda z. z z$$

are respectively equivalent to

$$(((\lambda z. z) ((\lambda x. z) x)) y) (\lambda x. x) \quad \lambda z. (z z)$$

Finally, the terms of the form $\lambda x. \lambda y. \dots$ are written as $\lambda x y. \dots$ for convenience.

With this simple setting, it is possible to capture the essence of computation. We know how programs look, but we do not know how programs compute. Intuitively, if we have a term of the form $(\lambda x. x) y$, then it should compute to y , but this step of computation is not declared in the syntax. How terms are computed or *evaluated* correspond to the *semantics of the calculus*. The semantics establishes what a program means and it can be formalized in a variety of ways. Throughout the document, we define the semantics of calculi through *operational semantics*. Operational semantics specify behavior by defining an abstract machine [20]. In essence, an operational semantics defines rules. These rules establish when a *step in computation* can be applied. The relation *step in computation* is defined as $t_1 \triangleright t_2$ and it defines that a term t_1 computes to t_2 . Before giving some examples, we have to introduce some definitions:

Definition 1 (Bound and free variables [20]). *A variable x is said to be bound if x is a sub-expression of $\lambda x. t$, and it is bound to the innermost enclosed syntactical lambda. A variable that is not bound, is said to be free.*

For example, in $\lambda x. x y z (\lambda x z. z x)$ the variable y is free, variable x is bound and the variable z is free in the outermost lambda and bound in the inner one. x has two different scopes. The x inside the inner lambda is not the same variable x as the outer lambda.

Definition 2 (Substitution [20]). *Given a term t and u , we define the substitution $t\{x := u\}$ where all free occurrences of x are replaced by u .*

With this definition we have that $(\lambda x. y x z (\lambda x z. z x))\{z := t\} \equiv \lambda x. y x t (\lambda x z. z x)$. Note that the inner z was not replaced because it is bound by the inner lambda.

Now we are able to define what a β -reduction is:

Definition 3 (β -reduction [20]). *Given a term with of the form $(\lambda x. t_1) t_2$, we say that it β -reduces to $t_1\{x := t_2\}$. The terms of the form $(\lambda x. t_1) t_2$ are called reducible expressions (redex), and the process of taking redex and applying the substitution in the previous way is known as β -reduction.*

Given the following term

$$(\lambda x. x) ((\lambda y. y) (\lambda z. z)) (\lambda z. z)$$

$$\frac{}{(\lambda x. t_1) t_2 \triangleright t_1 \{x := t_2\}} \quad \frac{t_1 \triangleright t_2}{t_1 t \triangleright t_2 t}$$

Figure 2.1: Example of operational semantics for extended λ -calculus

the term $(\lambda x. x) ((\lambda y. y) (\lambda z. z))$ is said to be the leftmost redex.

Note that β -reduction does not impose any condition on the terms. It only specifies when it can be applied. For example, a valid β -reduction of the term $\lambda x. (\lambda y. y) z$ is $\lambda x. z$. If this is a valid reduction or not depends on the semantics we are embedding in the calculus. More precisely, a reduction strategy establishes where it is possible to apply a reduction on a term [20].

- Full β -reduction: All redexes are reducible without order.
- Normal-order reduction: Leftmost, outermost redex expression is reduced first. All redex are reducible.
- Call-by-name reduction: Leftmost redex is reduced first until term is an abstraction. Arguments are not reduced. Reductions inside abstractions are forbidden.
- Call-by-value: Leftmost redex is reduced first until term is an abstraction. Arguments are then reduced by the same strategy. Then reduce the resulting redex. Reductions inside abstractions are forbidden as well.

As an example, we take the previous term $(\lambda x. x) ((\lambda y. y) (\lambda z. z)) (\lambda z. z)$. Remember that $(\lambda x. x) ((\lambda y. y) (\lambda z. z))$ is the leftmost redex. Under a call-by-name strategy, we have the following reduction:

$$(\lambda x. x) ((\lambda y. y) (\lambda z. z)) \triangleright x \{x := (\lambda y. y) (\lambda z. z)\} \equiv (\lambda y. y) (\lambda z. z)$$

because arguments are not reduced. On the other hand, the previous term reduces under a call-by-value strategy to:

$$(\lambda x. x) ((\lambda y. y) (\lambda z. z)) \triangleright (\lambda x. x) (y \{y := \lambda z. z\}) \equiv (\lambda x. x) (\lambda z. z)$$

In Figure 2.1 we have an example of operational semantics that defines a call-by-name strategy for the calculus. These rules establish the following reduction strategy: first evaluate the left term and substitute the right term in the left only if the left term is a lambda. Note that the first rule is just β -reduction.

Evaluating a term corresponds to keep applying reduction rules until no further rule applies. When no rules apply, the resulting term is in *normal form*.

Definition 4 (Normal form [20]). *A term t is said to be in normal form if no reduction rule applies.*

Here is an example of computation, assuming a call-by-value strategy (note the index to emphasis that β -reduction is being used):

$$(\lambda x. x) (\lambda y. y) z \triangleright_{\beta} (\lambda x. x) z \triangleright_{\beta} z$$

The variable z is the result of the computation because it is in normal form. Another example is

$$\begin{array}{c}
\frac{}{\text{if true then } t_1 \text{ else } t_2 \triangleright t_1} \quad \frac{}{\text{if false then } t_1 \text{ else } t_2 \triangleright t_2} \\
\frac{t \triangleright t'}{\text{if } t \text{ then } t_1 \text{ else } t_2 \triangleright \text{if } t' \text{ then } t_1 \text{ else } t_2} \\
\frac{\exists n_3, n_3 = n_1 + n_2}{n_1 + n_2 \triangleright n_3} \quad \frac{t_1 \triangleright t'_1}{t_1 + t_2 \triangleright t_1 + t_2} \quad \frac{t_2 \triangleright t'_2}{n + t_2 \triangleright n + t'_2}
\end{array}$$

Figure 2.2: Subset of rules of operational semantics for extended λ -calculus

$$(\lambda x. x x)(\lambda x. x x) \triangleright_{\beta} (\lambda x. x x)(\lambda x. x x) \triangleright_{\beta} \dots$$

We can see that the previous computation never stops reducing. This term is well known and defined as omega $\Omega := (\lambda x. x x)(\lambda x. x x)$, and it does not have a normal form. The fact that there exist non-terminating terms in the calculus has huge implications: being equivalent to Turing machines and the inconsistency as a logic.

The lambda calculus can also be seen as a simple programming language¹ with only function definition and application. While it is possible to encode the natural, booleans, and their operators (if, plus, etc.) in this simple language, another approach is to add them to the core. This extends the range of terms in the following way:

$$\begin{array}{l}
n := 1 \mid 2 \mid 3 \mid \dots \\
b := \text{true} \mid \text{false}
\end{array}$$

Here we define the booleans and naturals, and now we add them, plus their operators, to the calculus:

$$\begin{array}{l}
k := n \mid b \\
t := k \mid x \mid \lambda x. t \mid t t \mid t + t \mid t \times t \\
\text{if } t \text{ then } t \text{ else } t
\end{array}$$

Figure 2.2 presents an additional subset of operational semantics rules: to evaluate an *if* term we have to evaluate the conditional first in order to select the branch, and to evaluate a *sum* we have to evaluate the left term first and then the right term.

The additions to the core calculus expose some issues inherent to the λ -calculus. The following terms are valid in the language: $(\lambda x. x) 1$, $1 + 0$, **if true then false else true**, $\text{true} + 1$, (2 false) (note that the last one is an application). All these terms reach a normal form, but not all make *sense*: $\text{true} + 1$, (2 false) . Intuitively, a boolean and a number cannot be added, and a number cannot be applied to a boolean. In fact, if a term cannot be reduced, but it is not a valid term, then the term is *stuck*.

Definition 5 (Stuck term [20]). *A term t is said to be stuck if it is in normal form, but is not a value.*

The definition of *value* must be formalized by the calculus. For example, we can define that values in the calculus are: $\lambda x. x$, n and b .

¹The words *calculus* and *language* are used interchangeably

$$\begin{aligned}
B &:= \{\mathbf{nat}, \mathbf{bool}\} \\
T &:= \tau \mid T \rightarrow T \\
k &:= n \mid b \\
t &:= k \mid x \mid \lambda(x : T).t \mid t t \mid t + t \mid t \times t \\
&\quad \mathbf{if } t \mathbf{ then } t \mathbf{ else } t
\end{aligned}$$

Figure 2.3: Extended simple typed lambda calculus $\lambda_{E\rightarrow}$

All the previous issues motivated Church to define a variant of the λ -calculus: the typed lambda calculus.

2.2 Typed Lambda Calculus

The simply-typed lambda calculus [6] (λ_{\rightarrow}) is a variant of λ -calculus in which every abstraction specifies the type of its argument:

$$\begin{aligned}
T &:= \tau \mid T \rightarrow T \\
t &:= x \mid \lambda(x : T).t \mid t t
\end{aligned}$$

Where $\tau \in B$ with B being a set of base types.

Just as before, it is possible to add some primitives to the language just like in the untyped case (see Figure 2.3), which generates an extended simply typed lambda calculus ($\lambda_{E\rightarrow}$). Nevertheless, it is also possible to define the same *nonsensical* terms in this language. It seems that tagging the abstraction is useless. The missing key element is a *type system*. It is the type system that discards such terms.

2.2.1 Type System

A type system is set of rules, called typing rules, that ensures that a set of errors could not occur at execution time by doing a static analysis. This is done by applying the typing rules over the terms, assigning them types and ensuring that the terms are composed correctly. Thus, if a type system fails to type a term, the term probably has a flaw.

Type systems enforce properties. The rules that conform them are designed to establish properties over the terms of the language. For example, it is possible to define a type system over $\lambda_{E\rightarrow}$ such that only numbers can be added with the following rules:

$$\frac{n \in \mathbb{N}}{n : \mathbf{nat}} \qquad \frac{n_1 : \mathbf{nat} \quad n_2 : \mathbf{nat}}{n_1 + n_2 : \mathbf{nat}}$$

Type systems can enforce much more complex properties such as: concurrent programs without data races, keep track of effects, etc.

Figure 2.4 defines a type system for the extended typed lambda calculus. Note that in order to type check the terms, it is necessary to have *context* Γ . A context Γ is used to keep track of variable names and their types (similar to a list). For the particular case of the abstraction typing rule, it is necessary to assume that there exists a variable x with type A in the context plus the other variables and their types in order to check the type of the body of the abstraction. The definition of context is the following;

$$\begin{array}{c}
\frac{}{\Gamma \vdash \text{true} : \text{bool}} \quad \frac{}{\Gamma \vdash \text{false} : \text{bool}} \quad \frac{n \in \mathbb{N}}{\Gamma \vdash n : \text{nat}} \quad \frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \\
\\
\frac{\Gamma \vdash t_1 : \text{nat} \quad \Gamma \vdash t_3 : \text{nat}}{\Gamma \vdash t_1 + t_2 : \text{nat}} \quad \frac{\Gamma \vdash t_1 : \text{nat} \quad \Gamma \vdash t_3 : \text{nat}}{\Gamma \vdash t_1 \times t_2 : \text{nat}} \\
\\
\frac{\Gamma \vdash t_1 : \text{bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \\
\\
\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda(x : T_1).t : T_1 \rightarrow T_2} \quad \frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1 t_2 : T_2}
\end{array}$$

Figure 2.4: Type system for extended typed lambda calculus

$$\Gamma := \cdot \mid \Gamma, (x : T)$$

We define $\text{dom}(\Gamma)$ as $\{x \mid (x : T) \in \Gamma \text{ for some } T\}$. Also, due to α -conversion, we assume throughout this document that all variables are unique both in the context and terms.

The type of a term is derived as a tree. For example, the derivation tree of

$$\Gamma \vdash \text{if true then } (\lambda(x : \text{nat}).x) \ 1 \text{ else } n : \text{nat}$$

where $\Gamma := n : \text{nat}$ is

$$\frac{\frac{\frac{}{x : \text{nat} \in \Gamma, x : \text{nat}}}{\Gamma, x : \text{nat} \vdash x : \text{nat}} \quad \frac{1 \in \mathbb{N}}{\Gamma \vdash 1 : \text{nat}} \quad \frac{}{x : \text{nat} \in \Gamma}}{\Gamma \vdash (\lambda(x : \text{nat}).x) \ 1 : \text{nat}} \quad \frac{}{\Gamma \vdash \text{true} : \text{bool}}}{\Gamma \vdash \text{if true then } (\lambda(x : \text{nat}).x) \ 1 \text{ else } n : \text{nat}}$$

However, not all terms have a valid typing derivation: it is not possible to give a type to $\text{true} + 1$. When this happens, it is said that the term is *ill-typed*.

Note here that we have a similar syntax and the same semantics as the untyped lambda calculus, but on top we define a type system. Only well-typed terms can be executed in λ_{\rightarrow} .

2.2.2 Properties of the Simply-Typed Lambda Calculus

Working only with well-typed terms has some implications both as a programming language and as a logic. Before proceeding, we need to know what the *normalization property* is.

Definition 6 (Normalization [20]). *A calculus is said to be strongly normalizing if every valid term in the calculus is reducible to its normal form for every possible reduction sequence. A calculus is said to be weakly normalizing if for every term, there exists at least one normalizable sequence.*

The simply-typed lambda calculus is strongly normalizing with respect to β -reduction [24].

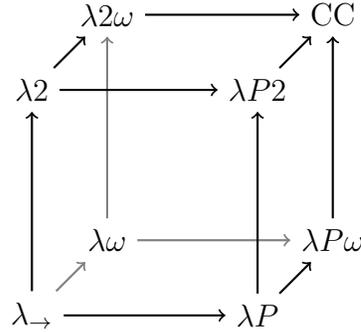
From a programming language perspective, recursion or self-referencing application is not supported anymore. This is due to the fact that well-typed terms in λ_{\rightarrow} must have a normal form and with recursion it is possible to define non-terminating terms. In general, it is not possible to give a type to non-terminating terms: Ω cannot be typed for example. As a result, this version of the lambda calculus is not Turing complete.

From a logic perspective, the calculus recovers logical consistency, as Church intended, and it is equivalent to first-order logic. Interestingly, because the calculus can be seen as a programming language, this establishes a connection between first-order logic and functional programming. In fact, this connection scales to logics in general and is called the *Curry-Howard correspondence* [12]: *propositions as types* [29], *proofs as programs*. The normalizing property plays a role in logic as well: this is the property that gives consistency, because a non terminating program could be used to prove the equivalent of the logical false \perp .

Logic	Calculus
Implication $P \supset Q$	Type $P \rightarrow Q$
Proof of proposition P	Term t with type P
Proposition P is provable	Type P is inhabited

2.2.3 Beyond the Simply-Typed Lambda Calculus

From a purely syntactic position, lambda terms are terms abstracted from other terms. It is important to know whether or not other kinds of abstraction exist. To explore this, Barendregt created the λ -cube [1]. In this system, it is possible to see how abstractions interact with each other and what calculus they form. Starting from the simply-typed lambda calculus, the calculi are ordered by inclusion. Each axis adds a specific abstraction. Here is a graphic representation of the λ -cube.



For our purpose, we present the adjacent vertices of λ_{\rightarrow} :

- Terms abstracted over types (λ^2): these terms are of the form $\Lambda X.t$. The type of these terms is $\forall X.T$ where X can be used as an identifier in T . These kinds of terms are known as parametric polymorphic terms. They can be instantiated with any type. The classic example is the identity term:

$$\Lambda X.\lambda(x : X).x : \forall X.X \rightarrow X$$

This term takes any type, any term of such type and returns such term.

- Types abstracted over types (λ^ω): these abstractions are of the form $\lambda(X :: K).T$. Here, K are *kinds* and they are the types of the types ($K ::= * \mid K \Rightarrow K$). Note that

in this case, the lambda is at the type level. It can be seen as an embedding of the simply-typed lambda calculus at the type level: base and function types have type $*$, and lambda on types has type $K \Rightarrow K'$ for some K and K' . This abstraction is known as type constructors. In order to show examples, we need to mix this calculus with $\lambda 2$. The type constructor for the list type can be

$$\text{List} := \lambda(A :: K).\forall X.X \rightarrow (A \rightarrow X \rightarrow X) \rightarrow X$$

Type constructors are closely related to church encodings [4].

- Types abstracted over terms (λP): these abstractions are at the type level. They are of the form $\Pi(x : T).T$ and represents functions where the parameter x can be used in T . The form of the type is known as dependent product and it is a generalization of the arrow function type. The arrow function type is a dependent product where the parameter is not used in the domain ($T_1 \rightarrow T_2 = \Pi(- : T_1), T_2$). This abstraction is known as dependent types because types can depend on terms

All the systems have been studied in [1]. The culmination of the λ -cube is the Calculus of Construction (CC) created by Coquand and Huet [8]. Here we have all abstractions: terms abstracted over types, types abstracted over types, and types abstracted over terms.

All the systems in λ -cube are known to be strongly normalizing. Also, all of them are related to some logic: $\lambda 2$, also known as **System F** introduced independently by [10, 21], is equivalent to second-order logic, for example.

2.2.4 Calculus of Constructions

In CC, the propositions (types) that can be stated are much more complex compared to the simply-typed lambda calculus. This is due to the combination of dependent types, type constructor and type quantification, which gives a very expressive framework to develop mathematics, via the Curry-Howard correspondence, whose proofs of propositions can be verified by a type system. Furthermore, because lambda calculus can also model programming languages, this gives a framework for establishing propositions over programs in the same language, which is the base of proof assistants.

2.3 Calculus of Inductive Constructions

In this section we present the calculus of inductive constructions (CIC) [16, 31] and how it can be used to develop mathematics. CIC is an extension of CC. The presentation we give is incremental, starting from a calculus without inductives, but equal in every other aspect. Such a calculus is the predicative calculus of constructions ($\text{CC}_{\omega+*}$), which is an extension of CC with a predicative sort²,

2.3.1 Terms

CIC was designed to not make a syntactic distinction between types and terms in the sense of simply-typed lambda calculus. In other words, types are a valid term in CIC. Types must

²This will be defined in a moment

$$A, B, C, D, I, N, M, R, S := \square \mid * \mid x \mid M N \mid \lambda(x : A). M \mid \Pi(x : A). B$$

Figure 2.5: Valid syntactic expression of $\text{CC}_{\omega+*}$

also type to something. The type of a type is called a *sort*

Definition 7 (Sort [16]). *Sorts are the types of types.*

Note that sorts can be seen as types, which means that they are also valid terms.

A first approach can be to define two sorts: $*$, read as Prop, which represents propositions and \square , read as Universe, which represents computational terms. Now, sorts are valid terms, meaning that they also must have a type. A naive definition of the typing rules for sorts could be:

$$\overline{* : \square} \qquad \overline{\square : \square}$$

However, the previous rules make the theory inconsistent, as proved by Girard [10], which is known as Girard’s paradox (the type theoretic Russell’s paradox of set theory). Stratifying \square avoids inconsistency.

More formally, sorts are defined in $\text{CC}_{\omega+*}$ as:

$$\mathcal{S} = \{*\} \cup \{\square_i \mid i \in \mathbb{N}\}$$

From now on, for every term s , we assume that $s \in \mathcal{S}$.

Lastly, we recall the definition of substitution defined for the untyped lambda calculus.

Definition 8 (Substitution). *Given terms M and N , then $M\{x := N\}$ represents the term M where all free occurrence of the variable x are replaced by the term N .*

To summarize, the valid syntactical expressions are: variables, lambda abstractions, applications, dependent products and sort. Figure 2.5 summarize the syntax.

2.3.2 Type System

We now have the valid syntactic expressions of the calculus. We proceed to define a type discipline over these expressions. This is done by simultaneously defining two judgments.

Before we can define the type system, we need to remember the notion of context. A context is used to keep track of variables and its type during the typing derivation of judgments. Therefore, a context is a list, where each element has the form of $x : A$.

Now, we are able to define the rules. The first one is $\Gamma \vdash M : A$ which establishes that the term M as type A in the context Γ and $\vdash \Gamma$, which establishes that the context is valid.

We can see the complete definition of the type system in Figure 2.6. We proceed to explain each of the rules:

- WF-EMPTY represents that an empty context is a valid context.

$$\begin{array}{c}
\text{WF-EMPTY} \quad \text{WF-HYP} \\
\frac{}{\vdash \cdot} \quad \frac{\Gamma \vdash A : s \quad x \notin \Gamma}{\vdash \Gamma, x : A} \\
\\
\text{TYPE} \quad \text{PROP} \quad \text{VAR} \\
\frac{\vdash \Gamma \quad i < j}{\Gamma \vdash \square_i : \square_j} \quad \frac{\vdash \Gamma}{\Gamma \vdash * : \square_i} \quad \frac{\vdash \Gamma \quad (x : A) \in \Gamma}{\Gamma \vdash x : A} \\
\\
\text{PROD-PROP-IMPR} \quad \text{PROD-TYPE-FORM} \\
\frac{\Gamma \vdash A : s \quad \Gamma, x : A \vdash B : *}{\Gamma \vdash \Pi(x : A). B : *} \quad \frac{\Gamma \vdash A : \square_i \quad \Gamma, x : A \vdash B : \square_j}{\Gamma \vdash \Pi(x : A). B : \square_{\max(i,j)}} \\
\\
\text{PROD-FORM} \\
\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : \square_i}{\Gamma \vdash \Pi(x : A). B : \square_i} \\
\\
\text{LAM} \quad \text{APP} \\
\frac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash \Pi(x : A). B : s}{\Gamma \vdash \lambda(x : A). M : \Pi(x : A). B} \quad \frac{\Gamma \vdash M : \Pi(x : A). B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B\{x := N\}} \\
\\
\text{CONV} \\
\frac{\Gamma \vdash M : B \quad \Gamma \vdash A : s \quad \Gamma \vdash A \equiv B}{\Gamma \vdash M : A}
\end{array}$$

Figure 2.6: Definition of $\text{CC}_{\omega+*}$ type system

- WF-HYP represents that a valid type can be assumed to be in a context and it generates a valid context.
- TYPE is the stratification presented previously to avoid Girard's paradox. It captures the essence that a lower universe can be embedded into a higher one. Luo's Extended Calculus of Construction [15] studied this stratification.
- PROP represents that a proposition can be embedded into any universe.
- VAR represents that if there exists a variable in the current context, then the variable with its corresponding type is a valid judgement.
- PROD-PROP-IMPR represents the impredicativity of $*$ when the result of a product is a proposition.
- PROD-TYPE-FORM represents that the higher universe prevails when forming a product.
- PROD-FORM represents that $*$ is not considered when it is an argument of a product.
- LAM represents the type of a lambda.
- APP represents the application of terms.
- CONV represents that a term can change its type to a new type if the new type is convertible with its current one. Essentially, types can be terms, and because terms

can be reduced, so can types. This rule captures that with the relation $\Gamma \vdash A \equiv B$. A more detail explanation will be given in Section 2.3.3.

2.3.3 Conversion Rules

Because types can depend on terms, it is necessary to allow *computations* at the type level. For example, the following types are essentially the same, but differ syntactically:

$$(\lambda(A : \square_{i+1}). A) \square_i \quad \square_i$$

We say that these types are intentionally equal, or convertible. The rules that establish that the previous types are convertible are the conversion rules and below we define all of them:

β -reduction: This rule was explained in the simple type lambda setting. Here, the properties of strong normalization and confluence are inherited, being CC_ω an extension of CC.

$$(\lambda(x : A). M) N \triangleright_\beta M\{x := N\}$$

η -expansion: This rule allows the identification of function terms with its expansion.

$$M \triangleright_\eta \lambda(x : A). M x$$

ι -reduction: This rule of conversion is related to inductive types. It will be defined in Section 2.3.4.

Convertibility

We define $\Gamma \vdash M \triangleright N$ as the compatible closure of $\triangleright_{\beta\iota\delta}$. We say that two terms M_1, M_2 are $\beta\iota\delta\eta$ -convertibles ($\Gamma \vdash M_1 \equiv M_2$) iff there exist some terms N_1, N_2 such that $\Gamma \vdash M_1 \triangleright \dots \triangleright N_1$ and $\Gamma \vdash M_2 \triangleright \dots \triangleright N_2$ and either N_1 and N_2 are identical, or they are convertible up to η -expansion

2.3.4 Inductive Types

Until now, we have defined the valid expression and the type system of $\text{CC}_{\omega+*}$. CIC is an extension of $\text{CC}_{\omega+*}$ with inductive types. However, the ability to add inductives to the language does not give any extra power to the calculus at all, because inductives can be defined by their Church encoding [11]. Nevertheless, in the latter approach it is impossible to define the induction principle for inductives [9], while in CIC is generated with a computational content. This gives more power to CIC in contrast to $\text{CC}_{\omega+*}$.

Inductive Introduction

We proceed to define the syntax for declaring inductives and the type checking constraints that must satisfy so that they can be added to the context. Note that in CIC it is possible to define mutual inductive types.

Definition 9 (Inductive Block [28]). *Inductive definitions are introduced by inductive blocks. Inductive blocks are of the form $\mathbf{Ind}_n \{ \Delta_I := \Delta_C \}$ where n represents the number of*

$$\begin{aligned}
& \mathbf{Ind}_0 \{ \text{empty} : \square_0 := \} \\
& \mathbf{Ind}_0 \{ \text{False} : * := \} \\
& \mathbf{Ind}_0 \{ \text{unit} : \square_0 := \text{tt} : \text{unit} \} \\
& \mathbf{Ind}_2 \{ \text{eq} : \Pi(A : \square_i)(x : A). A \rightarrow * := \text{eq_refl} : \Pi(A : \square_i)(x : A). \text{eq } A \ x \ x \} \\
& \mathbf{Ind}_2 \{ \text{ex} : \Pi(A : \square_i)(P : A \rightarrow *) . * := \text{ex_intro} : \Pi(A : \square_i)(P : A \rightarrow *) . \Pi(x : A). P \ x \rightarrow \text{ex } A \ P \} \\
& \mathbf{Ind}_0 \left\{ \begin{array}{l} \text{nat} : \square_0 := \text{0} : \text{nat} \\ \text{S} : \text{nat} \rightarrow \text{nat} \end{array} \right\} \\
& \mathbf{Ind}_1 \left\{ \begin{array}{l} \text{list} : \Pi(A : \square_i). \square_i := \text{nil} : \Pi(A : \square_i). \text{list } A \\ \text{cons} : \Pi(A : \square_i). A \rightarrow \text{list } A \rightarrow \text{list } A \end{array} \right\} \\
& \mathbf{Ind}_1 \left\{ \begin{array}{l} \text{vec} : \Pi(A : \square_i). \text{nat} \rightarrow \square_i := \text{vnil} : \Pi(A : \square_i). \text{vec } A \ 0 \\ \text{vcons} : \Pi(A : \square_i). \Pi(n : \text{nat}). A \rightarrow \text{vec } A \ n \rightarrow \text{vec } A \ (\text{S } n) \end{array} \right\} \\
& \mathbf{Ind}_0 \left\{ \begin{array}{l} \text{even} : \text{nat} \rightarrow \square_0 \quad \text{even0} : \text{even } 0 \\ \text{odd} : \text{nat} \rightarrow \square_0 \quad \text{evenS} : \Pi(n : \text{nat}). \text{odd } n \rightarrow \text{even } (\text{S } n) \\ \quad \quad \quad \text{oddS} : \Pi(n : \text{nat}). \text{even } n \rightarrow \text{odd } (\text{S } n) \end{array} \right\}
\end{aligned}$$

Figure 2.7: Examples of inductive types

$$\begin{aligned}
& \mathbf{Ind}_0 \{ \text{bad_ind}_1 : \square_0 := \text{bad}_1 : \text{bad_ind}_1 \rightarrow ((\text{unit} \rightarrow \text{bad_ind}_1) \rightarrow \text{unit}) \rightarrow \text{bad_ind}_1 \} \\
& \mathbf{Ind}_0 \{ \text{bad_ind}_2 : \square_i \rightarrow \square_i := \text{bad}_2 : \text{bad_ind}_2 \ \text{unit} \} \\
& \mathbf{Ind}_0 \{ \text{bad_ind}_3 : \square_i := \text{bad}_3 : \text{bad_ind}_1 \}
\end{aligned}$$

Figure 2.8: Examples of invalid inductive definitions

parameters of the types, Δ_I the names of the inductives associated with their types and Δ_C the constructors of the inductives under definition associated with their types.

Note that Δ_I and Δ_C are contexts. In addition, I is used to identify an inductive type and c represents constructors.

In Figure 2.7 and Figure 2.8 we see examples of (mutual) inductive definitions represented as inductive blocks. Note that the empty type does not have any constructor. Also, `eq` is the inductive types that represent equality in the type theory, so every time we see $N = M$, it is syntactic sugar for `eq A N M` where A is implicit from the context. Note that `eq` lives in $*$ which means that it represents a proposition. It is important to note that these examples only represent syntax and additional checks are required for them to be valid.

In order to maintain the logical consistency of the calculus, a restriction must be made over the structure of the inductive blocks. Thus, every time we declare an inductive block, we run a type check over the block before adding the inductive types and their constructors to the context. Before giving the typing rules for inductives, it is necessary to introduce some definitions.

Definition 10 (Arity of s [27]). A type A is said to be an arity of s , with $s \in \mathcal{S}$, if either

A is convertible to s or to a product $\Pi(x : B). A'$ with A' an arity of s .

For example, we have that $*$, $(\lambda(A : \square_{i+1}). A) \square_i$ and $\Pi A. A \rightarrow \square_i$ are arities of $*$, \square_i and \square_i , respectively. A valid type that is not an arity is $\Pi(A : s). A$. From the previous definition, we say that a type A is an arity if there exists some s such that A is an arity of s .

Definition 11 (Type constructor [27]). We say that C is a type constructor of I if:

- C is $I R_1 \dots R_n$
- C is $\Pi(x : A). C'$ where C' is a type constructor of I

From this we derive the concept of a constructor for a type I . If $(c : C) \in \Delta_C$, then we say that c is a constructor of I iff C is a type constructor of I . We see in Figure 2.7 that `even0` and `evenS` are constructors of `even`, and `oddS` is a constructor of `odd`. In the `nat` case, both `0` and `S` are constructors of `nat`.

The typing rules are declared in Figure 2.9 and they are added to the $CC_{\omega+*}$ ones (extracted from [27]).

Note that in `WF-IND` rule we have a set $\text{Constr}(\Delta_C, I)$, which represent the constructors of I in Δ_C , and a new predicate $\mathcal{I}_n(\Gamma, \Delta_I, \Delta_C)$ that verify the following conditions:

- All variables in Δ_I and Δ_C are distinct
- The first n arguments of all inductives and constructors of the block are the parameters. This means that there exists a sequence \vec{P} such that $\text{len}(\vec{P}) = n$ and for every $(x : A) \in \Delta_I, \Delta_C$ we have that $\Gamma \vdash A \equiv \Pi p : \vec{P}. B$ for some B . The notation $x : \vec{Y}$ represents $(x_1 : Y_1) \dots (x_k : Y_k)$ for some k where x_1, \dots, x_{i-1} can be used in Y_i .
- For every $I \in \text{dom}(\Delta_I)$, $(c : C) \in \Delta_C$ and $c \in \text{Constr}(\Delta_C, I)$ we have that $\Gamma \vdash C \equiv \Pi p : \vec{P}. \Pi u : \vec{U}. I \vec{p} \vec{u}$. Namely, the parameters of the constructor must be passed to the inductive type definition. It is said that parameters are parametric. Note that the rest of the arguments passed to the inductive (\vec{v}) are not necessarily the same as the constructor receives (\vec{x}).
- For every $(I : D) \in \Delta_I$ we have that:

$$\Gamma \vdash D \equiv \Pi p : \vec{P}. \Pi u : \vec{U}. s_I \quad \Gamma \vdash \Pi p : \vec{P}. \Pi u : \vec{U}. s_I : s'_I$$

Here \vec{U} is the arity (or indices) of the inductive and s_I its sort.

- For every $c \in \text{dom}(\Delta_C)$, exists $I \in \text{dom}(\Delta_I)$ such that $c \in \text{Constr}(\Delta_C, I)$.
- All inductive types in the block appear positively in the constructor. Positivity is a condition that ensures the consistency of the calculus when a new inductive type is being declared. Basically, it checks that no inductive being declared is used in a negative position. More information can be found in [27].

Coming back to Figure 2.7, we see that `bad_ind1`, `bad_ind2`, `bad_ind3` are not valid definition blocks. The first one because it does not satisfy the positivity condition, the second one because the constructor does not receive the parameters of the type and the third because `bad3` is not a constructor for `bad_ind3`

$$\begin{array}{c}
\text{WF-IND} \\
\frac{\mathcal{I}_n(\Gamma, \Delta_I, \Delta_C) \quad \Gamma \vdash D : s_I \quad \forall (I : D) \in \Delta_I}{\Gamma, \Delta_I \vdash C : s_I \quad \forall (c : C) \in \Delta_C \text{ if } c \in \text{Constr}(\Delta_C, I)} \\
\vdash \Gamma, \mathbf{Ind}_n \{ \Delta_I := \Delta_C \} \\
\\
\text{IND-TYPE} \\
\frac{\vdash \Gamma \quad \mathbf{Ind}_n \{ \Delta_I := \Delta_C \} \in \Gamma \quad (I : D) \in \Delta_I}{\Gamma \vdash I : D} \\
\\
\text{IND-CONSTR} \\
\frac{\vdash \Gamma \quad \mathbf{Ind}_n \{ \Delta_I := \Delta_C \} \in \Gamma \quad (c : C) \in \Delta_C}{\Gamma \vdash c : C}
\end{array}$$

Figure 2.9: Type judgments for inductive blocks

Inductive Elimination

Until now, we have established how to introduce new inductive types, which also gives a way to construct them (by applying the constructors). However, it is not possible to use (eliminate) them yet: check if a list is empty for example. In CIC, we have two mechanisms to eliminate inductive types: case block and induction principle. In this work, we focus on the induction principle, but we also take a look at the case block. First, we define the case block syntax because it offers a better intuition about what it means to eliminate an inductive type, and then we show that the induction principle can replace it, without losing expressivity, and more over, it gives more.

The main idea to eliminate a term whose type is an inductive one is to specify how it should behave for every constructor of the type, generating branches.

Definition 12 (Case block [27]). *Inductive elimination is represented by $\text{case}(\cdot; \cdot; \cdot \mid \dots \mid \cdot)$ where the first argument is the inductive term, the second one is a predicate over the term, and the third one represents the branches for each constructor.*

The predicate is a lambda whose arguments are the indices arguments of the inductive type plus the term of the case. The branches are lambdas that take the same arguments of corresponding positional constructor. The type of the block case corresponds to the application of the predicate with the indices of the inductive term of the block and the term itself.

Namely, given an inductive $I : \prod_{\vec{p} : \vec{P}} \prod_{u : U} s_I$, parameters $\vec{a} : \vec{P}$, indices $i : U\{\vec{p} := \vec{a}\}$ and a term M such as $\Gamma \vdash M : I \vec{a} i$, then the type of the predicate Q is

$$\prod_{u' : U\{\vec{p} := \vec{a}\}} \prod_{(x : I \vec{a} u')} R$$

and the type of the case block is $Q \vec{i} M$

Figure 2.10 shows the typing rule for case block. A more friendly representation of the previous case block is through pattern matching. The syntax is the following:

IND-ELIM

$$\begin{array}{c}
\vdash \Gamma \quad \mathbf{Ind}_n \{ \Delta_I := \Delta_C \} \in \Gamma \quad I : \Pi \vec{p} : \vec{P}. \Pi \vec{u} : \vec{U}. s_I \in \Delta_I \\
\{c_i\}_{i=1 \dots l} \in \mathbf{Constr}(\Delta_C, I) \quad (\Gamma \vdash c_i : \Pi \vec{p} : \vec{P}. \Pi \vec{x}_i : \vec{A}_i. I \vec{a} \vec{v}_i)_{i=1 \dots l} \\
\Gamma \vdash \vec{a} : \vec{P} \quad \Gamma \vdash i : U\{\vec{p} := \vec{a}\} \quad \Gamma \vdash M : I \vec{a} \vec{i} \quad [I|s_I|s_Q] \\
\Gamma \vdash Q : \Pi \vec{u}' : U\{\vec{p} := \vec{a}\}. \Pi(x : I \vec{a} \vec{u}'). s_Q \quad (\Gamma \vdash f_i : \Pi \vec{x}_i : \vec{A}_i\{\vec{p} := \vec{a}\}. Q \vec{v}_i (c_i \vec{p} \vec{x}_i))_{i=1 \dots l} \\
\hline
\Gamma \vdash \mathbf{case}(M; Q; f_1 | \dots | f_l) : Q \vec{i} M
\end{array}$$

Figure 2.10: Elimination of an inductive type

```

match inductive return predicate with
  constructor1 a11 ... a1k ⇒ result1
  ...
  constructorn an1 ... ank ⇒ resultn

```

This is a match block, but it represents the exact same case block defined above. Each line below the match statement is a branch of the inductive term and each argument \mathbf{a}_j^i on the right of the constructor is the variable binded on the corresponding branch in the case block.

The branches of the case block represent what should be done if the term of the case was built by the specific constructor. This means that it must be one branch for every constructor. Specifically, the branches are lambdas that take the non-parametric arguments of the respective constructor. The form of a case block can be represented as

$$\mathbf{case}(M; Q; f_1 | \dots | f_l)$$

Given the previous definition of case block, we are now capable of speaking about the other way of eliminating inductive type: the induction principle. The induction principle resembles the notion of mathematical induction: each form of constructing an object should conform to the given predicate and for each branch constructor, an induction hypothesis is available for arguments of the same type of the constructor that such an argument conforms to the predicate. Each inductive type is equipped with an induction principle that operates in a similar manner to the case block syntax: each principle must have a predicate and for each constructor, an abstraction must be provided that captures each argument of the constructor, but it also must have an argument for each recursive usage of the inductive type that proves the induction principle for that argument. In other words, given an inductive type I , and a constructor c of such type, whose type has the form

$$\Pi \vec{p} : \vec{P}. \Pi(\dots) (y : I \vec{v}') (\dots). I \vec{v}$$

then the corresponding branch of the inductive principle must have the form

$$\lambda(\dots) (y : I \vec{v}') (IH y : P \vec{v}' y) (\dots). (\dots)$$

The induction principle is an actual term of the calculus and has the form:

$$\mathbf{I_rect} \vec{p} Q f_1 \dots f_l \vec{i} M$$

Where f_1, \dots, f_l are the cases of branches. Its type is:

$$\vdash \mathbf{I_rect} : \Pi \overrightarrow{p} : \overrightarrow{P} (Q : \Pi \overrightarrow{u} : \overrightarrow{U} (v : I \overrightarrow{p} \overrightarrow{u}). s). \overrightarrow{C}_i \rightarrow \Pi i : \overrightarrow{U} (v : I \overrightarrow{p} \overrightarrow{u}). Q \overrightarrow{i} v$$

where \overrightarrow{C}_i conforms to the previous explanation of branches.

It is important to note that there exist two induction principles: one handles predicates that are an arity of $*$ ($\mathbf{I_rect}_*$) and the other handles predicates that are an arity of \square_i ($\mathbf{I_rect}_{\square_i}$). In practice, we can assume that the correct instance of induction principle is selected without declaring it. More details about induction principles can be found in [14].

Finally, induction principle terms can always be assumed, given that the related inductive type is valid.

It is now clear that the case block can be emulated by the induction principle. The case block forgets about the induction hypothesis in each branch of the induction principle.

Given that we now have inductive elimination, we can add an additional conversion rule that captures how inductives can be reduced. The conversion rule is ι -reduction and it is represented as \triangleright_ι . The rule allows the identification of a particular constructor with its corresponding branch

$$\mathbf{I_rect} \overrightarrow{p} Q f_1 \dots f_l \overrightarrow{i} (c_i \overrightarrow{p} \overrightarrow{u}) \triangleright_\iota f_i \overrightarrow{u}$$

We proceed to define some examples. The type of the induction principle of list is:

$$\begin{aligned} \vdash \mathbf{list_rect} : \Pi (A : \square) (Q : \mathbf{list} A \rightarrow \square_i). \\ Q \mathbf{nil} \rightarrow \\ (\Pi (a : A) (l : \mathbf{list} A). Q l \rightarrow Q (\mathbf{cons} a l)) \rightarrow \\ \Pi (l : \mathbf{list} A). Q l \end{aligned}$$

Now, the following term verifies if a list $l : \mathbf{list} A$ is empty:

$$\mathbf{list_rect} A (\lambda _ . \mathbf{bool}) \mathbf{true} (\lambda _ _ . \mathbf{false}) l$$

Here the predicate is homogeneous across all branches, so it does not need to bind its the arguments ($\lambda _ . M$ stands for the corresponding lambda in the context where no variable is binded). However, this is not always the case. Getting the head of a vector needs a smarter predicate. The type of the induction principle for vector is:

$$\begin{aligned} \vdash \mathbf{vect_rect} : \Pi (A : \square) (Q : \Pi (n : \mathbf{nat}). \mathbf{vec} A n \rightarrow \square_i). \\ Q \mathbf{0} \mathbf{vnil} \rightarrow \\ (\Pi (n : \mathbf{nat}) (a : A) (l : \mathbf{vec} A n). Q n l \rightarrow Q (\mathbf{S} n) (\mathbf{vcons} n a l)) \rightarrow \\ \Pi (n : \mathbf{nat}) (l : \mathbf{vec} A n). Q n l \end{aligned}$$

We want the following signature for the head function of vectors:

$$\mathbf{head} : \Pi (n : \mathbf{nat}) (A : \square_i) (v : \mathbf{vec} A (\mathbf{S} n)). A$$

Essentially, we are establishing through the type of the function that the vector parameter cannot be empty due to the $\mathbf{S} n$ length of the vector. Because there must be one branch per constructor, the naive definition of \mathbf{head} is not enough:

$$\lambda (a : \mathbf{nat}) (A : \square_i) (v : \mathbf{vec} A (\mathbf{S} n)). \mathbf{vect_rect} A (\lambda _ _ . A) \mathbf{tt} (\lambda _ (a : A) _ _ . a) (\mathbf{S} n) v$$

The previous definition has type:

$$(\lambda _ _ . A) (\mathbb{S} n) v \triangleright_{\beta} A$$

but the empty vector case has type `unit`, even though this case block is never applied to an empty vector. This is a sign that the predicate of the block needs to be refined: it depends on the indices of the inductive. Because the predicate can be any term, it is possible to perform a case analysis on the form of the indices of the type:

$$\begin{aligned} & \lambda(a : \mathbf{nat}) (A : \square_i) (v : \mathbf{vec} A (\mathbb{S} n)). \\ & \quad \mathbf{vec_rect} \\ & \quad \quad A \\ & \quad \quad (\lambda(n : \mathbf{nat}) _ _ . \mathbf{nat_rect} (\lambda _ _ . \square_i) \mathbf{unit} (\lambda _ _ . A) n) \\ & \quad \quad \mathbf{tt} \\ & \quad \quad (\lambda _ (a : A) _ _ . a) \\ & \quad \quad v \end{aligned}$$

The type of the previous terms correspond to apply predicate on the index and inductive:

$$\begin{aligned} & \Pi(a : \mathbf{nat}) (A : \square_i) (v : \mathbf{vec} A (\mathbb{S} n)). \\ & \quad (\lambda(n : \mathbf{nat}) _ _ . (\mathbf{nat_rect} (\lambda _ _ . \square_i) \mathbf{unit} A n)) (\mathbb{S} n) v \end{aligned}$$

Doing some β -reduction we get:

$$\Pi(a : \mathbf{nat}) (A : \square_i) (v : \mathbf{vec} A (\mathbb{S} n)). \mathbf{nat_rect} (\lambda _ _ . \square_i) \mathbf{unit} A (\mathbb{S} n)$$

Applying ι -reduction gives

$$\Pi(a : \mathbf{nat}) (A : \square_i) (v : \mathbf{vec} A (\mathbb{S} n)). A$$

which is the desired result for the type of `head`.

The list empty and vector head functions can be defined using case blocks. As shown, every case block can be replaced by the usage of the corresponding induction principle. However, the converse is not always true. For example, the length function that calculates the length of a list can be defined as

$$\lambda(A : \square) (l : \mathbf{list} A). \mathbf{list_rect} A (\lambda _ _ . \mathbf{nat}) 0 (\lambda _ _ . Pn. \mathbb{S} Pn) l$$

It is not possible to generate the previous term using only case blocks.

A final word about induction principles. Behind the scenes, induction principles are defined using case blocks and *fixpoints*. The latter is a primitive mechanism to allow primitive recursion, but it will not be explained here (see [26, 27]). This implies that the validity of an induction principle can be determined by the underlying case block and fixpoint. When we establish the validity of our work, we focus only on case blocks, but the elimination mechanism will be induction principles.

Inductive Elimination Restriction

CIC can be used as a mathematical framework to define propositions and to certify programs thanks to the sorts `*` and `\square_i` . Because of this, propositions should not influence computational terms, information cannot flow. In practice, it means that an inductive living in `*` cannot be eliminated into `\square_i` . Nevertheless, there exists an exception for this rule: *singleton elimination*

Definition 13 (Singleton elimination [26]). *Elimination of propositional terms into*

Term in CIC	Term in Coq
\square_i	<code>Type@{i}</code>
$*$	<code>Prop</code>
$\lambda(a : A). M$	<code>fun (a:A) => M</code>
$A \rightarrow B$	<code>A → B</code>
$\Pi(x : A). B$	<code>∀ (x:A), B</code>

Figure 2.11: Coq representations of CIC terms without inductives

computational terms is allowed if the inductive type being eliminated is either an empty definition or a single type with a single constructor where all the arguments are $$.*

As previously explained, every inductive type has two induction principles – one for each sort. The current restriction prohibits `I_rect` \square_i for inductive types that lives in $*$, except when they conform to the singleton elimination criteria.

2.4 Coq: The Proof Assistant

Coq [26] is a proof assistant that allows specification and development of software in the same environment. Coq implements CIC, which provides the tools for such environment. However, Coq uses CIC as the kernel and provides higher abstraction layers on top of it so it can be more user friendly. We proceed to explain how these two system relate.

First, it is necessary to define how the CIC terms are represented in Coq. It is important to note that the terms of Coq possess the same structure as the ones in CIC. Instead of providing the BNF syntax definition, we define them by a translation from CIC to Coq. This can be seen in Figure 2.11.

Typical Ambiguity

In CIC, the \square_i sort requires an explicit universe level, which is the index i . However, it is cumbersome to explicit each level. This is the reason Coq provides *typical ambiguity*. Typical ambiguity is the ability to use `Type` without an explicit level, and let the system keep a set of constraints in order to mantain consistency. The concept of typical ambiguity dates back to Russel’s type theory [22] and here it is used in a similar way.

Commands

Coq has an environment where it keeps all its declaration. Commands are statements that modify the environment. For example, `Definition` adds an identifier to the environment and it associates to a term, given that it is well-typed.

`Definition` definitions (A: `Type`) := A.

`Definition` definitions := `fun` (A:Type) => A.

The two previous examples are equal. `Inductive` generates the associated inductive block and it adds such block to the enviroment, given that it is well-typed.

Inductive unit: Type := tt: unit.

2.5 Adding Exceptions to CIC

In this section we are going to present one way of adding exceptions to CIC: *the exceptional type theory* [18]. We introduce the notion of syntactical translation and syntactical model, and the main properties of the exceptional type theory, its properties, drawbacks and extensions.

2.5.1 Models and Syntactic Translation

The calculi presented here are tools that formalize computations and even mathematics. However, calculi are presented in the same way it is presented here. It is natural then to question if they are *valid or correct*. One way to establish that a calculus is valid is to present a model of it. A model is a construction or translation of the terms in the language to another theory, usually set theory or category theory, where it is possible to establish the desired properties for the calculus. However, this way leaves two *disconnected* theories. Another approach is to derive properties in a syntactical manner from the calculus syntax and type system [3].

Another approach is to establish a syntactical translation from the calculus under study to another well-studied calculus. In [3], they took this approach, but the source and target theories were CIC. Also, because CIC can be used as a logic, this means that it is possible to establish properties of the source theory by working with the translated version.

Tabareau & Pédrot added exceptions to CIC using syntactic translations in [17] and [18]. The latter is the base of our work, so we dedicate the rest of the section to reviewing their work.

2.5.2 The Exceptional Type Theory

The exceptional type theory (ETT) [18] is a syntactical model that adds call-by-name exception to the language.

Call-by-name exceptions do not behave as expected compared to exceptions in mainstream languages. Exceptions (and computations in general) follow a call-by-value evaluation strategy, which means that arguments are evaluated before function execution. For example, in pseudo-code:

$$(\text{fun } x \Rightarrow 1) (\text{exception 'Fail'})$$

Under a call-by-value strategy, this term evaluates into an exception. In call-by-value, every argument is evaluated before the function body. As a consequence, if any argument evaluates into an exception, then the application evaluates into an exception.

$$(\text{fun } x \Rightarrow 1) (\text{exception 'Fail'}) \triangleright_{\beta} \text{exception 'Fail'}$$

On the other hand, the call-by-name strategy only evaluates the arguments when they are needed. In the previous example, the function application evaluates to 1 and the argument (which is an exception) is not evaluated because the function body does not use it.

`(fun x => 1) (exception 'Fail')` $\triangleright_\beta 1$

Coming back to ETT, the theory provides two new terms: an exception type $\mathbf{E} : \square_i$ and a raising function $\mathbf{raise} : \Pi(A : \square_i). \mathbf{E} \rightarrow A$. Because ETT is a syntactical model, the theory is generated by the translation $[\cdot]_\varepsilon$ where the following equation holds.

$$\Gamma \vdash_{\text{ETT}} M : A := [\Gamma]_\varepsilon \vdash [M]_\varepsilon : [A]_\varepsilon$$

The translation interprets a type A as a type A together with a default raising function $A_\phi : \mathbb{E} \rightarrow A$, where \mathbb{E} represents the target exception of \mathbf{E} . At the target level, the function \mathbf{raise} uses the raising function associated to each type.

The translation acts on every term, which means that \square_i must have this exceptional interpretation. The interpretation corresponds to:

$$\mathbf{Ind}_0 \left\{ \begin{array}{l} \text{type}_i : \square_{i+1} := \text{TypeVal}_i : \Pi(A : \square_i). (\mathbb{E} \rightarrow A) \rightarrow \text{type}_i \\ \text{TypeErr}_i : \mathbb{E} \rightarrow \text{type}_i \end{array} \right\}$$

where the translation is defined as:

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

Here, TypeErr_i is the raising function for \square_i . For $*$, a similar inductive is required (`prop`, `PropVal` and `PropErr`).

At the inductive level, it generates a new inductive with an additional constructor, which corresponds to the exceptional raising function for that type. For example, the translation of the list inductive type is:

$$\mathbf{Ind}_1 \left\{ \begin{array}{l} \text{nil}_\varepsilon : \Pi(A : \text{type}_i). \text{list}_\varepsilon A \\ \text{list}_\varepsilon : \Pi(A : \text{type}_i). \square_i := \text{cons}_\varepsilon : \Pi(A : \text{type}_i). \text{El}_i A \rightarrow \text{list}_\varepsilon A \rightarrow \text{list}_\varepsilon A \\ \text{list}_\phi : \Pi(A : \text{type}_i). \mathbb{E} \rightarrow \text{list}_\varepsilon A \end{array} \right\}$$

As a drawback, this translation has the problem that it makes ETT inconsistent as a logic, because every type can be inhabited by \mathbf{raise} . Even proofs.

2.5.3 The Parametric Exceptional Type Theory

The parametric exceptional type theory (PETT) [18] is another type theory which restricts the exceptions on ETT. The interpretation of this theory is also through a translation $[\cdot]_{\rho\varepsilon}$ which uses parametricity techniques [2] along the translation $[\cdot]_\varepsilon$. Essentially, this translation generates a proof that the term being translated is pure. In this theory, we have that:

$$\Gamma \vdash_{\text{PETT}} M : A := [\Gamma]_{\rho\varepsilon} \vdash [M]_{\rho\varepsilon} : [A]_{\rho\varepsilon} [M]_\varepsilon$$

In essence, the translation forbids any usage of the function \mathbf{raise} , which is not a pure term. However, it is possible that a term manages all its exceptions internally. The theory provides a way to generate a purity proof for such term.

This approach recovers logical consistency (exceptional proofs are not allowed), but it is not possible to enunciate exceptional theorems.

We covered the required background of our work. We presented the lambda calculus and its extensions. One of these extensions is the concept of a type system. We showed that type

systems, which can also be defined as type theories in our case, can be used to enunciate and prove theorems – the *Curry-Howard correspondence*. We presented CIC, a dependent type theory, that allows enunciating theorems that resembles mathematical theorems. Finally, we showed extensions of CIC that added exceptions, its properties and what is missing. In the next chapters, We present another extension of CIC that adds exceptions, but it addresses the problems of the previous extensions.

Chapter 3

Reasonably Exceptional Type Theory: RETT

3.1 Motivation

In the previous chapter, we presented a new type theory, ETT, which added exceptions to CIC [18]. This type theory has an exception type $\mathbf{E} : \square_0$ and a function $\mathbf{raise} : \Pi(A : \square_i). \mathbf{E} \rightarrow A$ which inhabits any type. The \mathbf{raise} function is the exception thrower of the calculus. For example, it can be used to generate the `head` and `tail` function for list assuming a term $e : \mathbf{E}$.

```
head := λ(A : □i) (l : list A). list_rect A (λ_ . A) (raise e) (λ(a : A) _ . A) l
tail := λ(A : □i) (l : list A).
      list_rect A (λ_ . list A) (raise e) (λ_ (l : list A) _ . A) l
```

It is also possible to define theorems for exceptional terms.

$$\vdash_{\text{ETT}} \Pi(A : \square_i) (l : \text{list } A). \text{length } l > 0 \rightarrow \text{tail } l \neq \text{raise } e$$

The previous propositions, which can be read as *applying tail over a non empty list cannot produce an exception* can be demonstrated by the following terms:

- $\lambda(A : \square_i) (l : \text{list } A). \dots$
- $\mathbf{raise} (\Pi(A : \square_i) (l : \text{list } A). \text{length } l > 0 \rightarrow \text{tail } l \neq \text{raise } e) e$

The first term is the actual proof of the property while the second is an exceptional proof (a proposition proved by an exception). Both terms are valid in ETT. In an attempt to control exceptions, [18] generated PETT. PETT requires a proof of purity for all terms, which implies that the second proof is not valid in this system. Unfortunately, the restrictions imposed by PETT are too strong: the theorem cannot be stated in PETT because it uses an exceptional term in the type.

$$\not\vdash_{\text{PETT}} \Pi(A : \square_i) (l : \text{list } A). \text{length } l > 0 \rightarrow \text{tail } l \neq \text{raise } e$$

The previous example shows two extreme when working with exceptions: one is too permissive and the other is too strict. We want something in the middle where it is possible to reason about exceptional terms but not allow exceptional proofs. The key point to note is

that we want exceptions in one hierarchy (\square_i) but forbid it in another ($*$), i.e., computational terms can be exceptional, but propositions cannot.

In this chapter we develop the *Reasonably Exceptional Type Theory* (RETT): a consistent exceptional type theory where it is possible to reason about exceptions, allowing only *pure* proofs. This implies that we can define the same theorem of ETT

$$\vdash_{\text{RETT}} \Pi(A : \square_i) (l : \text{list } A). \text{length } l > 0 \rightarrow \text{tail } l \neq \text{raise } e$$

but $\text{raise } (\Pi(A : \square_i) (l : \text{list } A). \text{length } l > 0 \rightarrow \text{tail } l \neq \text{raise } e) e$ cannot be used to prove such theorem.

First, we define the systems where we would be working. We then define the theory by a syntactic translation [3] by part: negative fragment and inductive types, and finally we would show some examples.

3.2 System Definition

The source and target theories are equal. It is named $\text{CC}_{\omega+*}^{\text{R}}$. The source theory has additionally two axioms, which corresponds to the exceptional type and the raising function. This system is similar to the one used in ETT. Here, we have two sorts: the hierarchical sort \square_i representing computational types and the impredicative sort $*$ representing propositions. We also have inductive types and singleton elimination. Finally, the only mechanism to eliminate inductive types is through induction principle. We summarize the system with both sorts and inductive types in Figure 3.1 and Figure 3.2.

Additionally, the theory provides two additional terms:

- $\mathbf{E} : \square_i$
- $\text{raise} : \Pi(A : \square_i). \mathbf{E} \rightarrow A$

3.3 Reasonably Exceptional Translation Definition

We already stated that we need to make a distinction between the computational sort \square_i and the propositional sort $*$. We need a way of establishing in which hierarchy a term lives. This can be accomplished by tagging every term with the sort it belongs to or carrying a context along the translation. In our work, we carry the context along the translation.

Remember that we want to forbid exception for $*$, which intuitively implies that the translation of $*$ should not be interpreted as an inductive type with a default raising function, because that would mean that it is possible to raise an exception over this sort. In the same line, we want to maintain singleton elimination, which means that the type translation from $*$ would be $*$. Also, no raising function should be defined for inductive types living in $*$.

These are the intuitions for defining the *reasonably exceptional translation* $[\cdot]_{\psi}^{\Gamma}$. The translation uses $[\cdot]_{\psi}^{\Gamma}$ at the term level and $[[\cdot]]_{\psi}^{\Gamma}$ at the type level.

We proceed by describing the required base structures of the translation, similar to the ones in ETT, then defining the translation for the negative fragment first and finally for the

$$\begin{array}{c}
A, B, N, M, R, S := \square_i \mid * \mid x \mid M N \mid \lambda(x : A). M \mid \Pi(x : A). B \\
\mathbf{Ind}_n \{ \Delta_I := \Delta_C \} \\
\Gamma, \Delta := \cdot \mid \Gamma, x : A \\
\begin{array}{c}
\text{TYPE} \qquad \qquad \text{PROP} \qquad \qquad \text{WEAKENING} \\
\frac{\vdash \Gamma \quad i < j}{\Gamma \vdash \square_i : \square_j} \quad \frac{\vdash \Gamma}{\Gamma \vdash * : \square_i} \quad \frac{\Gamma \vdash M : B \quad \Gamma \vdash A : s}{\Gamma, a : A \vdash M : B} \\
\text{TYPE-PROD} \qquad \qquad \text{PROP-PROD} \\
\frac{\Gamma \vdash A : \square_i \quad \Gamma, x : A \vdash B : \square_j}{\Gamma \vdash \Pi(x : A). B : \square_{\max(i,j)}} \quad \frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : \square_i}{\Gamma \vdash \Pi(x : A). B : \square_i} \\
\text{IMPRED} \qquad \qquad \text{CONV} \\
\frac{\Gamma \vdash A : s \quad \Gamma, x : A \vdash B : *}{\Gamma \vdash \Pi(x : A). B : *} \quad \frac{\Gamma \vdash M : B \quad \Gamma \vdash A : s \quad \Gamma \vdash A \equiv B}{\Gamma \vdash M : A} \\
\text{ABS} \qquad \qquad \text{APP} \\
\frac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash \Pi(x : A). B : s}{\Gamma \vdash \lambda(x : A). M : \Pi(x : A). B} \quad \frac{\Gamma \vdash M : \Pi(x : A). B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B\{x := N\}} \\
\text{IDEM} \qquad \qquad \text{WF-EMPTY} \qquad \qquad \text{WF-CONS} \\
\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \quad \frac{}{\vdash \cdot} \quad \frac{\Gamma \vdash A : s}{\vdash \Gamma, x : A} \\
\frac{}{\Gamma \vdash (\lambda(x : A). M) N \equiv M\{x := N\}} \quad \frac{}{\Gamma \vdash M \equiv \lambda(x : A). M x} \\
\frac{\Gamma \vdash M_1 \equiv M_2}{\Gamma \vdash M_1 N \equiv M_2 N} \quad \frac{\Gamma \vdash N_1 \equiv N_2}{\Gamma \vdash M N_1 \equiv M N_2}
\end{array}
\end{array}$$

Figure 3.1: $\text{CC}_{\omega+*}^{\text{R}}$ type system

inductive types. The soundness of the translation is proved for each part.

3.3.1 Base Structure

We present the base structures required by the translation. The base structures are similar to the ones in ETT

We assume in the target theory the following cumulative inductive type [28]:

$$\mathbf{Ind}_0 \left\{ \text{type}_i : \square_{i+1} := \frac{\text{TypeVal}_i : \Pi(A : \square_i). (\mathbb{E} \rightarrow A) \rightarrow \text{type}_i}{\text{TypeErr}_i : \mathbb{E} \rightarrow \text{type}_i} \right\}$$

with the following terms

$$\begin{array}{l}
- \Omega_i \quad : \mathbb{E} \rightarrow \square_i \\
- \omega_i \quad : \Pi(e : \mathbb{E}). \Omega_i e \\
- \text{type_elim}_{i,j} : \Pi(P : \text{type}_i \rightarrow \square_j). \\
\quad (\Pi(A : \square_i) (A_\phi : \mathbb{E} \rightarrow A). P (\text{TypeVal}_i A A_\phi)) \rightarrow \\
\quad (\Pi(e : \mathbb{E}). P (\text{TypeErr}_i e)) \rightarrow \Pi(T : \text{type}_i). P T
\end{array}$$

and computation rules

$$\begin{array}{c}
\text{WF-IND} \\
\frac{\mathcal{I}_n(\Gamma, \Delta_I, \Delta_C) \quad \Gamma \vdash D : s_I \quad \forall (I : D) \in \Delta_I}{\Gamma, \Delta_I \vdash C : s_I \quad \forall (c : C) \in \Delta_C \text{ if } c \in \text{Constr}(\Delta_C, I)} \\
\hline
\vdash \Gamma, \mathbf{Ind}_n \{ \Delta_I := \Delta_C \} \\
\\
\text{IND-TYPE} \\
\frac{\vdash \Gamma \quad \mathbf{Ind}_n \{ \Delta_I := \Delta_C \} \in \Gamma \quad (I : D) \in \Delta_I}{\Gamma \vdash I : D} \\
\\
\text{IND-CONSTR} \\
\frac{\vdash \Gamma \quad \mathbf{Ind}_n \{ \Delta_I := \Delta_C \} \in \Gamma \quad (c : C) \in \Delta_C}{\Gamma \vdash c : C}
\end{array}$$

Figure 3.2: $\text{CC}_{\omega+*}^{\text{R}}$ type system for inductive types

$$\begin{array}{l}
\text{type_elim}_{i,j} P p_v p_\phi (\text{TypeVal}_i A A_\phi) \equiv p_v A A_\phi \\
\text{type_elim}_{i,j} P p_v p_\phi (\text{TypeErr}_i e) \equiv p_\phi e
\end{array}$$

Also, we define the following functions

$$\begin{array}{l}
\text{El}_i : \text{type}_i \rightarrow \square_i \\
:= \lambda(A : \text{type}_i). \\
\quad \text{type_elim}_{i,i} (\lambda(T : \text{type}_i). \square_i) (\lambda(A_0 : \square_i) (A_\phi : \mathbb{E} \rightarrow A_0). A_0) \Omega_i A \\
\text{Err}_i : \Pi(A : \text{type}_i). \mathbb{E} \rightarrow \text{El}_i A \\
:= \lambda(A : \text{type}_i) (e : \mathbb{E}). \\
\quad \text{type_elim}_{i,i} \text{El}_i (\lambda(A_0 : \square_i) (A_\phi : \mathbb{E} \rightarrow A_\phi). A_\phi e) \omega A
\end{array}$$

3.3.2 Negative Fragment

We start by defining the translation for the negative fragment. The translation works at the type level and at the term level. As stated before, we want the type translation of $*$ be equal to $*$ – we want the following equivalence to hold for every context.

$$\llbracket * \rrbracket_\psi^\Gamma \equiv *$$

This would avoid the exceptional terms on propositions and maintain the semantics of propositions. Another point to take into consideration is that $\Gamma \vdash * : \square_i$, which implies that at the term level should be wrapped over a `TypeVal` because $\Gamma \vdash \llbracket * \rrbracket_\psi^\Gamma : \text{type}_i$. This means that we have to provide the raising function for $*$. We simply choose a degenerate function which cannot be inhabited: $\lambda(e : \mathbb{E}). \Pi(A : *). A$. The body of the function has type $*$, but the type is equivalent to `False`. It means that we cannot generate a function with that type. These requirements drive the translation.

We define the translation in Figure 3.3. Note that this translation is equal to the exceptional translation if we restrict to the computational sort. The translation needs to know to which sort the term being translated belongs, so it passes the context to its subterms, and even extend the context for lambdas and binders. Lastly, the exceptional raiser is only defined for the computational hierarchy and only used in this context which a posteriori is valid and accomplishes our goals.

We can see in Figure 3.4 the differences between the exceptional translation and the reasonably exceptional translation both at term level and type level. Note that in the reasonably

$$\begin{array}{l}
\llbracket \square_i \rrbracket_{\psi}^{\Gamma} \quad := \text{TypeVal}_i \text{ type}_i \text{ TypeErr}_i \\
\llbracket * \rrbracket_{\psi}^{\Gamma} \quad := \text{TypeVal}_i * (\lambda(e : \mathbb{E}). \Pi(A : *) . A) \\
\llbracket x \rrbracket_{\psi}^{\Gamma} \quad := x \\
\llbracket \lambda(x : A) . M \rrbracket_{\psi}^{\Gamma} \quad := \lambda(x : \llbracket A \rrbracket_{\psi}^{\Gamma}) . \llbracket M \rrbracket_{\psi}^{\Gamma, x:A} \\
\llbracket M N \rrbracket_{\psi}^{\Gamma} \quad := \llbracket M \rrbracket_{\psi}^{\Gamma} \llbracket N \rrbracket_{\psi}^{\Gamma} \\
\llbracket \Pi(x : A) . M \rrbracket_{\psi}^{\Gamma} \quad := \begin{cases} \Pi(x : \llbracket A \rrbracket_{\psi}^{\Gamma}) . \llbracket B \rrbracket_{\psi}^{\Gamma, x:A} & \Gamma \vdash \Pi(x : A) . B : * \\ \text{TypeVal}_i & \sim \\ \Pi(x : \llbracket A \rrbracket_{\psi}^{\Gamma}) . \llbracket B \rrbracket_{\psi}^{\Gamma, x:A} & \\ \lambda(e : \mathbb{E}) (x : \llbracket A \rrbracket_{\psi}^{\Gamma}) . \llbracket B \rrbracket_{\psi\phi}^{\Gamma, x:A} e & \end{cases} \\
\llbracket A \rrbracket_{\psi\phi}^{\Gamma} \quad := \text{Err}_i \llbracket A \rrbracket_{\psi}^{\Gamma} \\
\llbracket A \rrbracket_{\psi}^{\Gamma} \quad := \begin{cases} \llbracket A \rrbracket_{\psi}^{\Gamma} & \Gamma \vdash A : * \\ \text{El}_i \llbracket A \rrbracket_{\psi}^{\Gamma} \sim & \end{cases} \\
\hline
\llbracket \cdot \rrbracket_{\psi} \quad := \cdot \\
\llbracket \Gamma, x : A \rrbracket_{\psi} \quad := \llbracket \Gamma \rrbracket_{\psi}, x : \llbracket A \rrbracket_{\psi}^{\Gamma}
\end{array}$$

Figure 3.3: Reasonably Exceptional Translation Definition

exceptional translation, every term that lives in \square_i is wrapped under a `TypeVal` but not if it lives in `*`. In the last entry of the b table, only the universe is translated and is equal at the term level and type level.

We now prove some lemmas that will be required later.

Lemma 1. *The following equivalences hold:*

- $\llbracket \square_i \rrbracket_{\psi}^{\Gamma} \equiv \text{type}_i$
- $\llbracket * \rrbracket_{\psi}^{\Gamma} \equiv *$
- $\llbracket \Pi(x : A) . B \rrbracket_{\psi}^{\Gamma} \equiv \Pi(x : \llbracket A \rrbracket_{\psi}^{\Gamma}) . \llbracket B \rrbracket_{\psi}^{\Gamma, x:A}$

Proof. By unfolding definitions:

- $\begin{aligned} \llbracket \square_i \rrbracket_{\psi}^{\Gamma} &\equiv \text{El}_i \llbracket \square_i \rrbracket_{\psi}^{\Gamma} \\ &\equiv \text{El}_i (\text{TypeVal}_i \text{ type}_i \text{ TypeErr}_i) \\ &\equiv \text{type}_i \end{aligned}$
- $\begin{aligned} \llbracket * \rrbracket_{\psi}^{\Gamma} &\equiv \text{El}_i \llbracket * \rrbracket_{\psi}^{\Gamma} \\ &\equiv \text{El}_i (\text{TypeVal}_i * (\lambda(e : \mathbb{E}). \Pi(A : *) . A)) \\ &\equiv * \end{aligned}$

- This case depends on $\Gamma \vdash \Pi(x : A) . B : *$ being derivable. If it is, then:

$$\begin{aligned}
\llbracket \Pi(x : A) . B \rrbracket_{\psi}^{\Gamma} &\equiv \llbracket \Pi(x : A) . B \rrbracket_{\psi}^{\Gamma} \\
&\equiv \Pi(x : \llbracket A \rrbracket_{\psi}^{\Gamma}) . \llbracket B \rrbracket_{\psi}^{\Gamma, x:A}
\end{aligned}$$

	$[\cdot]_\varepsilon$	$\llbracket \cdot \rrbracket_\varepsilon$
\square_i	$\text{TypeVal}_i \text{ type}_i \text{ TypeErr}_i$	type_i
$*$	$\text{TypeVal}_i \text{ prop}_i \text{ TypeErr}_i$	prop_i
$\square_i \rightarrow \square_j$	TypeVal_i $(\text{type}_i \rightarrow \text{type}_j)$ $(\lambda(e : \mathbb{E}) (t : \text{type}_i). \text{TypeErr}_j e)$	$\text{type}_i \rightarrow \text{type}_j$
$\square_i \rightarrow *$	TypeVal_i $(\text{type}_i \rightarrow \text{prop}_i)$ $(\lambda(e : \mathbb{E}) (t : \text{type}_i). \text{PropErr } e)$	$\text{type}_i \rightarrow \text{prop}_i$
$\Pi(A : *).$ $\square_i \rightarrow A$	PropVal $(\Pi(A : \text{prop}_i). \text{type}_i \rightarrow A)$ $(\lambda(e : \mathbb{E}) (A : \text{prop}_o) (_ : \text{type}_i). \text{PropErr } e)$	$\Pi(A : \text{prop}_i). \text{type}_i \rightarrow A$

(a) Translation for $[\cdot]_\varepsilon / \llbracket \cdot \rrbracket_\varepsilon$

	$[\cdot]_\psi^\Gamma$	$\llbracket \cdot \rrbracket_\psi^\Gamma$
\square_i	$\text{TypeVal}_i \text{ type}_i \text{ TypeErr}_i$	type_i
$*$	$\text{TypeVal}_i * (\Pi(A : *). A)$	$*$
$\square_i \rightarrow \square_j$	TypeVal_i $(\text{type}_i \rightarrow \text{type}_j)$ $(\lambda(e : \mathbb{E}) (t : \text{type}_i). \text{TypeErr}_j e)$	$\text{type}_i \rightarrow \text{type}_j$
$\square_i \rightarrow *$	TypeVal_i $(\text{type}_i \rightarrow *)$ $(\lambda(e : \mathbb{E}) (t : \text{type}_i). \Pi(A : *). A)$	$\text{type}_i \rightarrow *$
$\Pi(A : *). \square_i \rightarrow A$	$\Pi(A : *). \square_i \rightarrow A$	$\Pi(A : *). \square_i \rightarrow A$

(b) Translation for $[\cdot]_\psi^\Gamma / \llbracket \cdot \rrbracket_\psi^\Gamma$

Figure 3.4: Difference between the translations

Otherwise, we have:

$$\begin{aligned}
\llbracket \Pi(x : A). B \rrbracket_\psi^\Gamma &\equiv \text{El}_i \llbracket \Pi(x : A). B \rrbracket_\psi^\Gamma \\
&\equiv \text{El}_i (\text{TypeVal}_i (\Pi(x : \llbracket A \rrbracket_\psi^\Gamma). \llbracket B \rrbracket_\psi^{\Gamma, x:a}) (\lambda(e : \mathbb{E}). \Pi(x : \llbracket A \rrbracket_\psi^\Gamma). \llbracket B \rrbracket_\psi^{\Gamma, x:a} e)) \\
&\equiv \Pi(x : \llbracket A \rrbracket_\psi^\Gamma). \llbracket B \rrbracket_\psi^{\Gamma, x:a}
\end{aligned}$$

□

Lemma 2. *Given a context Γ , a term M and a sort s such that $\Gamma \vdash \llbracket M \rrbracket_\psi^\Gamma : \llbracket s \rrbracket_\psi^\Gamma$, then $\Gamma \vdash \llbracket M \rrbracket_\psi^\Gamma : s$.*

Proof. It is important to note that the next proof is developed under $\text{CC}_{\omega+*}^R$ where it is valid to apply typing rules if the conditions are met. By a case analysis over s .

Case $s := \square_i$:

After reduction, we have to prove $\Gamma \vdash \mathbf{El}_i [M]_\psi^\Gamma : \square_i$ given $\Gamma \vdash [M]_\psi^\Gamma : \mathbf{type}_i$. Because \mathbf{El}_i has type $\mathbf{type}_i \rightarrow \square_i$, the result is obtained by an application of the **APP** rule.

Case $s := *$:

After reduction, we have to prove $\Gamma \vdash \llbracket M \rrbracket_\psi^\Gamma : *$ given $\Gamma \vdash [M]_\psi^\Gamma : *$, which is trivial. □

Lemma 3 (Substitution). *Given a context Γ , terms M , N and A , such that $\Gamma \vdash N : A$. We have:*

$$[M\{x := N\}]_\psi^\Gamma \equiv [M]_\psi^{\Gamma, x:A} \{x := [N]_\psi^\Gamma\}$$

Proof. By induction on M over a generalized context

Case $M := x, M := y$:

These cases are trivial.

Case $M := \square_i, M := *$:

These cases are also trivial because x is not involved in the translated term due to α -rename.

Case $M := M_1 M_2$:

We have $[(M_1 M_2)\{x := N\}]_\psi^\Gamma$. By unfolding the definition of substitution and the translation we get:

$$\begin{aligned} [(M_1 M_2)\{x := N\}]_\psi^\Gamma &\equiv [M_1\{x := N\} M_2\{x := N\}]_\psi^\Gamma \\ &\equiv [M_1\{x := N\}]_\psi^\Gamma [M_2\{x := N\}]_\psi^\Gamma \end{aligned}$$

It is possible to apply the induction hypothesis over M_1 and M_2

$$\begin{aligned} ([M_1]_\psi^{\Gamma, x:A} \{x := [N]_\psi^\Gamma\}) ([M_2]_\psi^{\Gamma, x:A} \{x := [N]_\psi^\Gamma\}) &\equiv ([M_1]_\psi^{\Gamma, x:A} [M_2]_\psi^{\Gamma, x:A}) \{x := [N]_\psi^\Gamma\} \\ &\equiv [M_1 M_2]_\psi^{\Gamma, x:A} \{x := [N]_\psi^\Gamma\} \end{aligned}$$

which is the required result.

Case $M := \lambda(x : B). M_1$:

Note that in this case the binder is the same being substituted, which means that no substitution should be applied in the body. By unfolding the definition of substitution and the translation, we get:

$$[\lambda(x : B). M_1]_\psi^\Gamma \equiv \lambda(x : \llbracket B\{x := N\} \rrbracket_\psi^\Gamma). [M_1]_\psi^{\Gamma, x:B}$$

It is necessary to do a case analysis over $\Gamma \vdash B\{x := N\} : s$, with s a sort, to unfold $\llbracket B\{x := N\} \rrbracket_\psi^\Gamma$ which could give either:

- $[B\{x := N\}]_\psi^\Gamma$
- $\mathbf{El}_i [B\{x := N\}]_\psi^\Gamma$

In both cases, it follows directly that $\llbracket B\{x := N\} \rrbracket_\psi^\Gamma \equiv \llbracket B \rrbracket_\psi^{\Gamma, x:A} \{x := [N]_\psi^\Gamma\}$ by using the induction hypothesis and unfolding/folding. Then, the conclusion can be established.

Case $M := \lambda(y : B). M_1$:

The binder in this case is distinct from the one being substituted. Nevertheless, it is possible

to derive by a similar reasoning to the previous case that substitution is for the type. It remains to continue in the body of the lambda:

$$\lambda(y : \llbracket B \rrbracket_{\psi}^{\Gamma, x:A} \{x := \llbracket N \rrbracket_{\psi}^{\Gamma}\}) . \llbracket M_1 \{x := N\} \rrbracket_{\psi}^{\Gamma, y:B}$$

Here, it is possible to apply the induction hypothesis over $\llbracket M \{x := N\} \rrbracket_{\psi}^{\Gamma, y:B}$ because the context can be instantiated arbitrarily,

$$\llbracket M \rrbracket_{\psi}^{\Gamma, y:B, x:A} \{x := \llbracket N \rrbracket_{\psi}^{\Gamma, y:B}\}$$

Additionally, we can assume that $y \notin \text{FreeVar}(N)$ due to α -rename, which implies

$$\llbracket N \rrbracket_{\psi}^{\Gamma, y:B} \equiv \llbracket N \rrbracket_{\psi}^{\Gamma}$$

This is valid because the context is used at two places in the translation where it is valid to swap for a context without an unbound variable. We also have:

$$\llbracket M \rrbracket_{\psi}^{\Gamma, y:B, x:A} \equiv \llbracket M \rrbracket_{\psi}^{\Gamma, x:A, y:B}$$

because $y \notin A$ due to α -rename. Note that we swap the variables in the context. Then, by folding the translation we obtain:

$$\llbracket \lambda(y : B) . M \rrbracket_{\psi}^{\Gamma, x:A} \{x := \llbracket N \rrbracket_{\psi}^{\Gamma}\}$$

Case $M := \Pi(y : A_1) . B$:

For this case, we do case analysis on the sort s to which $\Pi(y : A_1) . B$ types. It is important to note that this case is similar to the abstraction case. This means that we can unfold substitution over the binder without loss of generality.

Subcase $s := *$:

We have after reduction

$$\Pi(y : \llbracket A_1 \{x := N\} \rrbracket_{\psi}^{\Gamma}). \llbracket B \{x := N\} \rrbracket_{\psi}^{\Gamma, y:A_1}$$

we can conclude that $\Gamma, y : A_1 \vdash B : *$ by inversion over $\Gamma \vdash \Pi(y : A_1) . B : *$. We can reduce the term with this hypothesis to

$$\Pi(y : \llbracket A_1 \{x := N\} \rrbracket_{\psi}^{\Gamma}). \llbracket B \{x := N\} \rrbracket_{\psi}^{\Gamma, y:A_1}$$

In this scenario, the remaining of the subcase is equal to the lambda case.

Subcase $s := \square_i *$:

In this case, we have after reduction

$$\text{TypeVal}_i M_1 M_2$$

with

- $M_1 := \Pi(y : \llbracket A_1 \{x := N\} \rrbracket_{\psi}^{\Gamma}). \llbracket B \{x := N\} \rrbracket_{\psi}^{\Gamma, y:A_1}$
- $M_2 := \lambda(e : \mathbb{E}) (y : \llbracket A_1 \{x := N\} \rrbracket_{\psi}^{\Gamma}). \llbracket B \{x := N\} \rrbracket_{\psi\phi}^{\Gamma, y:A_1}$

Both cases were solved previously by an unfolding, induction hypothesis application and folding again.

□

Lemma 4 (Conversion). *Given a context Γ and terms M and N , if $\Gamma \vdash M \equiv N$ then $\llbracket \Gamma \rrbracket_\psi \vdash \llbracket M \rrbracket_\psi^\Gamma \equiv \llbracket N \rrbracket_\psi^\Gamma$*

Proof. By induction over the typing derivation with a generalized Δ .

Case β -reduction:

Here we have

- $M := (\lambda(x : A). M_1) N_1$
- $N := M\{x := N_1\}$

Unfolding the translation and applying Lemma 3:

- $M := (\lambda(x : \llbracket A \rrbracket_\psi^\Gamma). \llbracket M_1 \rrbracket_\psi^{\Gamma, x:A}) \llbracket N_1 \rrbracket_\psi^\Gamma$
- $N := \llbracket M \rrbracket_\psi^{\Gamma, x:A} \{x := \llbracket N_1 \rrbracket_\psi^\Gamma\}$

The result can be established by a β -reduction.

Case η -expansion:

Here we have

- $M := M_1$
- $N := \lambda(x : A). M x$

Unfolding the translation

- $M := \llbracket M_1 \rrbracket_\psi^\Gamma$
- $N := \lambda(x : \llbracket A \rrbracket_\psi^\Gamma). \llbracket M \rrbracket_\psi^{\Gamma, x:A} x$

Same as before, it is possible to establish the conversion between the two terms by an η -expansion.

Case App:

Without loss of generality, we can set the conversion to be established for the left term of the application. The right case is similar. We have

- $M := M_1 N'$
- $N := M_2 N'$

with $\Gamma \vdash M_1 \equiv M_2$. Unfolding the translation we get:

- $\llbracket M_1 \rrbracket_\psi^\Gamma \llbracket N' \rrbracket_\psi^\Gamma$
- $\llbracket M_2 \rrbracket_\psi^\Gamma \llbracket N' \rrbracket_\psi^\Gamma$

Applying the induction hypothesis allows the conclusion.

□

Theorem 5 (Reasonably Exceptional Translation Soundness). *Given Γ , M and A such that $\Gamma \vdash M : A$, then we have $\llbracket \Gamma \rrbracket_\psi \vdash \llbracket M \rrbracket_\psi^\Gamma : \llbracket A \rrbracket_\psi^\Gamma$.*

Proof. By an induction over the typing derivation. We reduce the terms in most of the cases.

Case TYPE:

We need to prove $\llbracket \Gamma \rrbracket_\psi \vdash \text{TypeVal}_i \text{ type}_i \text{ TypeErr}_i : \text{type}_i$. We know that TypeVal_i has type $\Pi(A : \text{type}_{i'}) . (\mathbb{E} \rightarrow A) \rightarrow \text{type}_{i'}$ and type_i has type \square_j with $i < j$. By a double application of APP, we can conclude the result unifying j with i' .

Case PROP:

We need to prove $\llbracket \Gamma \rrbracket_\psi \vdash \text{TypeVal}_i * (\Pi(A : *) . A) : \text{type}_i$. This is similar to the previous case.

Case WEAKENING:

We need to prove $\llbracket \Gamma \rrbracket_\psi, a : \llbracket A \rrbracket_\psi^\Gamma \vdash \llbracket M \rrbracket_\psi^\Gamma : \llbracket B \rrbracket_\psi^\Gamma$. By the induction hypothesis we have $\llbracket \Gamma \rrbracket_\psi \vdash \llbracket M \rrbracket_\psi^\Gamma : \llbracket B \rrbracket_\psi^\Gamma$ and $\llbracket \Gamma \rrbracket_\psi \vdash \llbracket A \rrbracket_\psi^\Gamma : \llbracket s' \rrbracket_\psi^\Gamma$. It only remains to show $\llbracket \Gamma \rrbracket_\psi \vdash \llbracket A \rrbracket_\psi^\Gamma : s'$. for some s' . This can be Lemma 2 and the result can be established by WEAKENING.

Case ABS:

We need to prove

$$\llbracket \Gamma \rrbracket_\psi \vdash \lambda(x : \llbracket A \rrbracket_\psi^\Gamma) . \llbracket M \rrbracket_\psi^{\Gamma, x:A} : \Pi(x : \llbracket A \rrbracket_\psi^\Gamma) . \llbracket B \rrbracket_\psi^{\Gamma, x:A}$$

The induction hypothesis gives us

- $\llbracket \Gamma \rrbracket_\psi, x : \llbracket A \rrbracket_\psi^\Gamma \vdash \llbracket M \rrbracket_\psi^{\Gamma, x:A} : \llbracket B \rrbracket_\psi^{\Gamma, x:A}$
- $\llbracket \Gamma \rrbracket_\psi \vdash \llbracket \Pi(x : A) . B \rrbracket_\psi^\Gamma : \llbracket s \rrbracket_\psi^\Gamma$

We can derive $\llbracket \Gamma \rrbracket_\psi \vdash \llbracket \Pi(x : A) . B \rrbracket_\psi^\Gamma : s$ by Lemma 2. Noticing that

$$\Gamma \vdash \llbracket \Pi(x : A) . B \rrbracket_\psi^\Gamma \equiv \Pi(x : \llbracket A \rrbracket_\psi^\Gamma) . \llbracket B \rrbracket_\psi^{\Gamma, x:A}$$

the previous hypotheses allow us to obtain the result by an application of ABS rule.

Case TYPE-PROD:

We need to prove

$$\llbracket \Gamma \rrbracket_\psi \vdash \text{TypeVal}_{\max(i,j)} (\Pi(x : \llbracket A \rrbracket_\psi^\Gamma) . \llbracket B \rrbracket_\psi^{\Gamma, x:A}) (\lambda(e : \mathbb{E}) (x : \llbracket A \rrbracket_\psi^\Gamma) . \llbracket B \rrbracket_\psi^{\Gamma, x:A} e) : \text{type}_{\max(i,j)}$$

We are able to conclude this case if the first argument of $\text{TypeVal}_{\max(i,j)}$ types to $\text{type}_{\max(i,j)}$ and the second to $\mathbb{E} \rightarrow \Pi(x : \llbracket A \rrbracket_\psi^\Gamma) . \llbracket B \rrbracket_\psi^{\Gamma, x:A}$ by a double application of APP. We proceed to check the first argument.

By the induction hypothesis we know that:

- $\llbracket \Gamma \rrbracket_\psi \vdash \llbracket A \rrbracket_\psi^\Gamma : \text{type}_i$
- $\llbracket \Gamma \rrbracket_\psi, x : \llbracket A \rrbracket_\psi^\Gamma \vdash \llbracket B \rrbracket_\psi^{\Gamma, x:A} : \text{type}_j$

We can derive:

- $\llbracket \Gamma \rrbracket_\psi \vdash \llbracket A \rrbracket_\psi^\Gamma : \square_i$
- $\llbracket \Gamma \rrbracket_\psi, x : \llbracket A \rrbracket_\psi^\Gamma \vdash \llbracket B \rrbracket_\psi^{\Gamma, x:A} : \square_j$

by Lemma 2. By TYPE-PROD we get

$$\llbracket \Gamma \rrbracket_\psi \vdash \Pi(x : \llbracket A \rrbracket_\psi^\Gamma) . \llbracket B \rrbracket_\psi^{\Gamma, x:A} : \square_{\max(i,j)}$$

It remains to show that

$$\llbracket \Gamma \rrbracket_\psi \vdash \lambda(e : \mathbb{E}) (x : \llbracket A \rrbracket_\psi^\Gamma). \llbracket B \rrbracket_{\psi\phi}^{\Gamma, x:A} e : \mathbb{E} \rightarrow \Pi(x : \llbracket A \rrbracket_\psi^\Gamma). \llbracket B \rrbracket_\psi^{\Gamma, x:A}$$

In order to establish this result, we need that

- $\llbracket \Gamma \rrbracket_\psi \vdash \mathbb{E} \rightarrow \Pi(x : \llbracket A \rrbracket_\psi^\Gamma). \llbracket B \rrbracket_\psi^{\Gamma, x:A} : \square_{\max(i,j)}$
- $\llbracket \Gamma \rrbracket_\psi, e : \mathbb{E} \vdash \lambda(x : \llbracket A \rrbracket_\psi^\Gamma). \llbracket B \rrbracket_{\psi\phi}^{\Gamma, x:A} e : \Pi(x : \llbracket A \rrbracket_\psi^\Gamma). \llbracket B \rrbracket_\psi^{\Gamma, x:A}$

The first case is demonstrated by an application of **TYPE-PROD** because $\forall \Gamma', \Gamma' \vdash \mathbb{E} : \square_0$ and $\llbracket \Gamma \rrbracket_\psi \vdash \Pi(x : \llbracket A \rrbracket_\psi^\Gamma). \llbracket B \rrbracket_\psi^{\Gamma, x:A} : \square_{\max(i,j)}$ was already established and extending the context in this case can be done as result of the calculus. For the second case, it can be applied an **ABS** and will only be missing (after reduction)

$$\Gamma, e : \mathbb{E}, x : \llbracket A \rrbracket_\psi^\Gamma \vdash \mathbf{Err}_i \llbracket B \rrbracket_\psi^{\Gamma, x:A} e : \mathbf{El}_i \llbracket B \rrbracket_\psi^{\Gamma, x:A}$$

By a double application of **APP** can be established. Most of the hypothesis are already defined for a context without $e : \mathbb{E}$. This variable can be added and permuted to its position by properties of the calculus.

Case PROP-PROD:

This case is similar to the previous one with the difference that now we have $\llbracket \Gamma \rrbracket_\psi \vdash \llbracket A \rrbracket_\psi^\Gamma : *$ instead of $\llbracket \Gamma \rrbracket_\psi \vdash \llbracket A \rrbracket_\psi^\Gamma : \square_i$. This holds because $\llbracket A \rrbracket_\psi^\Gamma \equiv \llbracket A \rrbracket_\psi^\Gamma$ and $\llbracket \Gamma \rrbracket_\psi \vdash \llbracket A \rrbracket_\psi^\Gamma : *$ and holds by the induction hypothesis.

The rest of the proof remains almost unchanged.

Case IMPRED:

In this case we need to prove

$$\llbracket \Gamma \rrbracket_\psi \vdash \lambda(x : \llbracket A \rrbracket_\psi^\Gamma). \llbracket M \rrbracket_\psi^{\Gamma, x:A} : *$$

because $\Gamma \vdash \lambda(x : A). M : *$. By the induction hypothesis we have

- $\llbracket \Gamma \rrbracket_\psi \vdash \llbracket A \rrbracket_\psi^\Gamma : \llbracket s \rrbracket_\psi^\Gamma$
- $\llbracket \Gamma \rrbracket_\psi, x : \llbracket A \rrbracket_\psi^\Gamma \vdash \llbracket B \rrbracket_\psi^{\Gamma, x:A} : *$

It is possible to derive $\llbracket \Gamma \rrbracket_\psi \vdash \llbracket A \rrbracket_\psi^\Gamma : s$ by Lemma 2. With the previous hypothesis, the desired result is obtained by a direct application of **IMPRED** rule.

Case APP:

Here we need to prove

$$\llbracket \Gamma \rrbracket_\psi \vdash \llbracket M N \rrbracket_\psi^\Gamma : \llbracket B\{x := N\} \rrbracket_\psi^\Gamma$$

By an unfolding the translation and Lemma 3

$$\llbracket \Gamma \rrbracket_\psi \vdash \llbracket M \rrbracket_\psi^\Gamma \llbracket N \rrbracket_\psi^\Gamma : \llbracket B \rrbracket_\psi^{\Gamma, x:A} \{x := \llbracket N \rrbracket_\psi^\Gamma\}$$

We also have as induction hypothesis

- $\llbracket \Gamma \rrbracket_\psi \vdash \llbracket M \rrbracket_\psi^\Gamma : \Pi(x : \llbracket A \rrbracket_\psi^\Gamma). \llbracket B \rrbracket_\psi^{\Gamma, x:A}$
- $\llbracket \Gamma \rrbracket_\psi \vdash \llbracket N \rrbracket_\psi^\Gamma : \llbracket A \rrbracket_\psi^\Gamma$

which allows us to conclude by the APP rule.

Case CONV:

We need to prove

$$\llbracket \Gamma \rrbracket_\psi \vdash [M]_\psi^\Gamma : [A]_\psi^\Gamma$$

The induction gives us

- $\llbracket \Gamma \rrbracket_\psi \vdash [M]_\psi^\Gamma : [B]_\psi^\Gamma$
- $\llbracket \Gamma \rrbracket_\psi \vdash [A]_\psi^\Gamma : [s]_\psi^\Gamma$

We also have

- $\llbracket \Gamma \rrbracket_\psi \vdash [A]_\psi^\Gamma \equiv [B]_\psi^\Gamma$

by Lemma 4. We know that $\llbracket \Gamma \rrbracket_\psi \vdash [A]_\psi^\Gamma : s$ by Lemma 2. We need to check

$$\llbracket \Gamma \rrbracket_\psi \vdash [A]_\psi^\Gamma \equiv [B]_\psi^\Gamma$$

There are four cases:

- $\llbracket \Gamma \rrbracket_\psi \vdash \text{El}_i [A]_\psi^\Gamma \equiv \text{El}_i [B]_\psi^\Gamma$
- $\llbracket \Gamma \rrbracket_\psi \vdash [A]_\psi^\Gamma \equiv [B]_\psi^\Gamma$
- $\llbracket \Gamma \rrbracket_\psi \vdash [A]_\psi^\Gamma \equiv \text{El}_i [B]_\psi^\Gamma$
- $\llbracket \Gamma \rrbracket_\psi \vdash \text{El}_i [A]_\psi^\Gamma \equiv [B]_\psi^\Gamma$

The first and second cases are allowed because of the hypothesis and conversion rules. The third and fourth are impossibles because if the terms have those forms, it is because one belongs to $*$ and the other to \square_i , and those sorts are not convertible in the source and target theories. Now it is possible to apply CONV, which gives the result.

Case IDEM:

We need to prove

$$\llbracket \Gamma \rrbracket_\psi, x : [A]_\psi^\Gamma \vdash x : [A]_\psi^\Gamma$$

given

$$\llbracket \Gamma \rrbracket_\psi \vdash [A]_\psi^\Gamma : [s]_\psi^\Gamma$$

by Lemma 2 we have $\llbracket \Gamma \rrbracket_\psi \vdash [A]_\psi^\Gamma : s$, which allows us to conclude by IDEM rule.

Case WF-EMPTY:

Trivial.

Case WF-CONS:

We need to prove $\llbracket \Gamma \rrbracket_\psi, x : [A]_\psi^\Gamma$. By the induction hypothesis we get $\llbracket \Gamma \rrbracket_\psi \vdash [A]_\psi^\Gamma : [\square_i]_\psi^\Gamma$.

By Lemma 2 we get $\llbracket \Gamma \rrbracket_\psi \vdash [A]_\psi^\Gamma : \square_i$ and the result can be derived by WF-CONS

□

3.3.3 Inductive Types

We have defined the translation for the negative fragment. We proceed to define it over inductive types: this means that we need to translate induction blocks (introduction) and generate the corresponding induction principle. Remember that we explained that induction principles are composed of a case block and a fixpoint. We argue that to in order to define the translation of induction principles, it is only required to show the translation of case blocks. Note that this can be seen as a meta-property of $\text{CC}_{\omega+*}^{\text{R}}$, because case blocks are not exposed in the calculus.

Definition 14 (Inductive Reasonably Exceptional Translation). *Let $\mathbf{Ind}_n \{ \Delta_I := \Delta_C \}$ be a mutual inductive block. We assume that:*

- $\forall (I : D) \in \Delta_I, \exists \vec{p} : \vec{P} \vec{u} : \vec{U}, D \equiv \Pi \vec{p} : \vec{P}. \Pi \vec{u} : \vec{U}. s_I$
- $\forall (c : C) \in \Delta_C, \exists I \in \text{dom}(\Delta_I), \forall c \in \text{Constr}(\Delta_C, I), \exists \vec{p} : \vec{P} \vec{a} : \vec{A},$
 $C \equiv \Pi \vec{p} : \vec{P}. \Pi \vec{a} : \vec{A}. I_c \vec{p} \vec{a}$

Then, we can define:

- $\Delta_{I_\psi} := \left\{ I_\psi : \Pi [\vec{p} : \vec{P}]_\psi^\Gamma. \Pi [\vec{u} : \vec{U}]_\psi^{\Gamma, \vec{p} : \vec{P}}. s_I \mid (I : D) \in \Delta_I \right\}$
- $\Delta'_{C_\psi} := \left\{ c_\psi : \Pi [\vec{p} : \vec{P}]_\psi^\Gamma. \Pi [\vec{a} : \vec{A}]_\psi^{\Gamma, \vec{p} : \vec{P}}. [I_c]_\psi^{\Gamma'} \vec{p} [\vec{a}]_\psi^{\Gamma'} \mid (c : C) \in \Delta_C \wedge \Gamma' := \Gamma, \vec{p} : \vec{P}, \vec{a} : \vec{A} \right\}$
- $\Delta_{I_\phi} := \left\{ I_\phi : \Pi [\vec{p} : \vec{P}]_\psi^\Gamma. \Pi [\vec{u} : \vec{U}]_\psi^{\Gamma, \vec{p} : \vec{P}}. \mathbb{E} \rightarrow [I]_\psi^\Gamma \vec{p} \vec{u} \mid (I : D) \in \Delta_I \right\}$
- $\Delta_{C_\psi} := \Delta'_{C_\psi} \cup \Delta_{I_\phi}$

setting locally in the translation:

- $[I]_\psi^\Gamma := I_\psi$
- $[[I \vec{p} \vec{u}]]_\psi^\Gamma := I_\psi [\vec{p}]_\psi^\Gamma [\vec{u}]_\psi^\Gamma$

With these definitions, we define the translation of a mutual inductive block as

$$[\mathbf{Ind}_n \{ \Delta_I := \Delta_C \}]_\psi^\Gamma := \mathbf{Ind}_n \{ \Delta_{I_\psi} := \Delta_{C_\psi} \}$$

and extend the translation with:

- $[I]_\psi^\Gamma := \begin{cases} \lambda [\vec{p} : \vec{P}]_\psi^\Gamma [\vec{u} : \vec{U}]_\psi^{\Gamma, \vec{p} : \vec{P}}. I_\psi \vec{p} \vec{u} & \text{If } I \text{ belongs to } * \\ \lambda [\vec{p} : \vec{P}]_\psi^\Gamma [\vec{u} : \vec{U}]_\psi^{\Gamma, \vec{p} : \vec{P}}. \text{TypeVal}_i (I_\psi \vec{p} \vec{u}) (I_\phi \vec{p} \vec{u}) & \text{If } I \text{ belongs } \square_i \end{cases}$
- $[c]_\psi^\Gamma := c_\psi$

We will see examples in the next section. We proceed to define the extension for case block elimination.

Definition 15 (Case Reasonably Exceptional Translation). *Given a context Γ , terms \vec{p} and \vec{u} , an inductive term M such that $\Gamma \vdash M : I \vec{p} \vec{u}$, a predicate $Q := \lambda (i : \vec{U}) (x : I \vec{p} i). R$,*

branch terms \vec{f}_j and a case block term $\text{case}(M; Q; \vec{f}_j)$ such that $\Gamma \vdash \text{case}(M; Q; \vec{f}_j) : Q \vec{u} M$. Defining:

$$\begin{aligned} Q_\psi &:= \lambda \llbracket i : U \rrbracket_\psi^\Gamma (x : [I]_\psi^\Gamma [\vec{p}]_\psi^\Gamma \vec{i}). \llbracket R \rrbracket_\psi^{\Gamma, i:U, x:I} \vec{p} \vec{i} \\ f_{\psi\phi} &:= \lambda \llbracket i : U \rrbracket_\psi^\Gamma (e : \mathbb{E}). [R]_{\psi\phi}^{\Gamma, i:U} \{x := [I]_\psi^\Gamma [\vec{p}]_\psi^\Gamma \vec{i}\} e \end{aligned}$$

Then, we define the translation of case blocks as:

$$[\text{case}(M; Q; \vec{f}_j)]_\psi^\Gamma := \begin{cases} \text{case}([M]_\psi^\Gamma; Q_\psi; [\vec{f}_j]_\psi^\Gamma \mid f_{\psi\phi}) & \text{If } I \text{ and } Q \text{ belongs to } \square_i \\ \text{case}([M]_\psi^\Gamma; Q_\psi; [\vec{f}_j]_\psi^\Gamma) & \text{If } I \text{ belongs to } * \end{cases}$$

Note that the previous definition is not complete: elimination from \square_i to $*$ is not allowed. The insight for this definition is that terms belonging to \square_i can be exceptions in RETT, but terms belonging to $*$ cannot. Thus, case block over terms in \square_i cannot handle its exceptional constructor. From now on, we assume that for every case block, it is possible to define its translation.

We now proceed to prove the soundness of the extension of the reasonably exceptional translation that enables it to handle inductive types.

Theorem 6 (Inductive Extension Reasonably Exceptional Translation Soundness). *Extending the translation for inductive introduction (Definition 14) and inductive elimination (Definition 15) preserves typing soundness.*

Proof. We need to prove the related typing judgement for each extension in order to prove the preservation of typing soundness. By induction on the typing derivation, analyzing only the new cases.

Case Inductive Introduction:

We have three judgements related to inductive introduction

Subcase WF-IND:

The well foundedness of the translation is a direct consequence of Theorem 5. All the typing requirements are derived and the pertinence to the corresponding sets are granted by construction. The interesting case is $\mathcal{I}_n(\cdot, \cdot, \cdot)$. Remember that this predicate checks syntactic constructions of blocks declarations. The translation preserves these syntactic constructions and all conditions can be satisfied by construction.

Subcase IND-TYPE and IND-CONSTR:

These cases are derived by the induction hypothesis and translation definition.

Case Inductive Elimination:

The only way to eliminate inductive is through each induction principle. As explained, in order to derive translation of an induction principle, it is only required to check it for the underlying case block. In this case we only need to show IND-ELIM (see Figure 2.10) is respected under the translation. This is a direct application of the induction hypothesis, translation definition and context extension. The interesting case is when eliminating from \square_i to \square_i because it adds the exceptional constructor on the case block. This is granted by construction because the exceptional branch takes the same arguments as the constructor and the bindings are the same of Q . This implies that this branch generates the expected

typing for a branch in the case block.

□

Now that we have the soundness of inductive introductions, we are able to process the corresponding induction principle by a simple translation of the term. Thus, we have the exceptional induction principle available in the target theory, with some restrictions. Elimination from inductives in \square_i to $*$ is not allowed, because the underlying case block does not support it. Finally, elimination from \square_i to \square_i simply re-raise the exception to the corresponding type. This means that exceptions flow in the system.

Nevertheless, the translation is able to generate a *catch induction principle* for each sort. This induction principle is equal to the corresponding induction principle, but it contains an additional case for the exceptional constructor. This means that we can eliminate computational inductive terms into propositions through the catch induction principle. For example, here we can see the case for `nat` (`nat_catch`)

$$\begin{aligned} & \Pi(P : \mathbf{nat} \rightarrow s). \\ & P \mathbf{0} \rightarrow \\ & (\Pi(n : A). P n \rightarrow P (\mathbf{S} n)) \rightarrow \\ & (\Pi(e : \mathbb{E}). P (\mathbf{nat}_\psi e)) \rightarrow \\ & \Pi(n : \mathbf{nat}). P n \end{aligned}$$

RETT also enjoys additional reduction rules. Assume an inductive I and its induction principle `I_rect`. Then, the first additional reduction rule is

$$\mathbf{I_rect} \vec{p} P f_1 \dots f_l \vec{i} (\mathbf{raise} (I \vec{p} \vec{i}) e) \equiv \mathbf{raise} (P (\mathbf{raise} (I \vec{p} \vec{i}) e)) e$$

This rule establishes that exceptions flow across the theory. The other rule is for the catch induction principle:

$$\mathbf{I_catch} \vec{p} P f_1 \dots f_l f_\phi \vec{i} (\mathbf{raise} (I \vec{p} \vec{i}) e) \equiv f_\phi$$

This rule establishes that the catch induction principle reduces to the exceptional branch if an exception is applied.

Finally, the generated type theory RETT preserves singleton elimination:

Corollary 7. *If a inductive type I meets the criteria for singleton elimination in the source theory, then the translated inductive type meets the criteria for singleton elimination*

Proof. This is direct from the equivalence $\llbracket * \rrbracket_\psi^\Gamma \equiv *$ and the fact that the translation does not add an additional constructor for propositional inductive types. □

3.4 Examples

In this section, we show examples of the previous translation and how it can be used. We first need to realize the two terms of the theory: $\mathbf{E} : \square_i$ and $\mathbf{raise} : \Pi(A : \square_i). \mathbf{E} \rightarrow A$:

- $\llbracket \mathbf{E} \rrbracket_\psi^\Gamma := \mathbf{TypeVal}_i \mathbb{E} (\lambda(e : \mathbb{E}). e)$
- $\llbracket \mathbf{raise} \rrbracket_\psi^\Gamma := \lambda(A : \mathbf{type}_i) (e : \mathbb{E}). \mathbf{Err}_i A e$

In other words, the exceptional type is simply a reification of the underlying exceptional type, and the raising function simply uses the raising function associated to the type to inhabit any type in the theory.

Now, we can see how the translation works at the induction level. For computational inductives, we can see that it adds a default constructor, which is the default raising function for that type:

$$\mathbf{Ind}_0 \left\{ \begin{array}{l} \mathbf{0}_{\psi} : \mathbf{nat}_{\psi} \\ \mathbf{nat}_{\psi} : \square_0 := \mathbf{S}_{\psi} : \mathbf{nat}_{\psi} \rightarrow \mathbf{nat}_{\psi} \\ \mathbf{nat}_{\phi} : \mathbb{E} \rightarrow \mathbf{nat}_{\psi} \end{array} \right\}$$

$$\mathbf{Ind}_1 \left\{ \begin{array}{l} \mathbf{nil}_{\psi} : \Pi(A : \mathbf{type}_i). \mathbf{list}_{\psi} A \\ \mathbf{list}_{\psi} : \Pi(A : \mathbf{type}_i). \square_i := \mathbf{cons}_{\psi} : \Pi(A : \mathbf{type}_i). \mathbf{El}_i A \rightarrow \mathbf{list}_{\psi} A \rightarrow \mathbf{list}_{\psi} A \\ \mathbf{list}_{\phi} : \Pi(A : \mathbf{type}_i). \mathbb{E} \rightarrow \mathbf{nat}_{\psi} \end{array} \right\}$$

In contrast, the translation does not add the default raising function for propositional inductives, but it translates the inner computational types:

$$\mathbf{Ind}_0 \{ \mathbf{False}_{\psi} : * := \}$$

$$\mathbf{Ind}_2 \{ \mathbf{eq}_{\psi} : \Pi(A : \mathbf{type}_i)(x : \mathbf{El}_i A). \square_i := \mathbf{eq_refl}_{\psi} : \Pi(A : \mathbf{type}_i)(x : \mathbf{El}_i A). \mathbf{eq}_{\psi} A x x \}$$

Note that the translation of **False** has no constructors. Here, we can see why **raise** cannot be applied over propositions. It cannot rescue the raising function.

In this chapter, we defined the Reasonably Exceptional Type Theory (RETT): a consistent type theory with exceptions. We presented the mechanisms the theory provides for working with exceptions such as induction principles. We established the soundness of the theory through a syntactic translation into CIC, which provides good meta-theoretical properties. Nonetheless, there is room for improvement: trivial theorems of CIC are not valid in RETT. In the next chapter, we present examples to illustrate this and we propose an extension over RETT to overcome this gap.

Chapter 4

Controlling Exceptions: A Modality over RETT

4.1 Motivation

The previous chapter describes RETT: a type theory with exceptions. We also saw some applications of the type theory, showing that it is possible to raise/catch exceptions on programs, but it is also possible to reason about exceptions while maintaining logical consistency.

Now, the theory treats every computational term as a possible exceptional term. This happens because every computational type is interpreted as a type with a raising function. The theory takes this in consideration and it provides the catch induction principle. This principle allows considering the case when the term is an exception. However, if the result belongs to a computational type, then it is also seen as a term that can be an exception, even though the exception was already managed. For example, the following empty function uses the catch induction for list to always provide a *pure* value, even if the list is an exception.

```
 $\lambda(A : \square_i) (l : \text{list } A). \text{list\_catch } (\lambda\_ \_ . \text{bool}) \text{ true } (\lambda a \_ \_ . \text{false}) (\lambda \_ \_ . \text{false}) l$ 
```

The previous function returns `true` only if the list is empty. When it is not or it is an exception, returns `false`. It is not possible that the result of this function returns an exception, but the theory still interprets the result as if it can be.

Some theorems, which are provable in CIC, do not hold in RETT. For example

$$\vdash \Pi(n : \text{nat}). n \geq 0$$

This theorem states that every natural number is bigger than zero. On the other hand, we have in RETT:

$$\not\vdash_{\text{RETT}} \Pi(n : \text{nat}). n \geq 0$$

This theorem is not valid in RETT because natural numbers are computational types. The theory sees a natural number as a possible exception and the proposition \geq do not relate an exceptional number with a concrete number.

We would like to have a way to enforce purity (restrict exceptions) on terms:

$$\mathbf{check}_{\rho\psi}(\Pi(x : A). B) \equiv \mathbf{check}_{\rho\psi}(B) \quad \mathbf{check}_{\rho\psi}(M N) \equiv \mathbf{check}_{\rho\psi}(M) \quad \mathbf{check}_{\rho\psi}(A) \equiv A$$

Figure 4.1: Checker function definition

$$\vdash_{\text{RETT}} \Pi(n : \text{nat}). \mathbf{param} \ n \rightarrow n \geq 0$$

Here, we are using `param n` to say that the number n is pure, which enables proving that $n \geq 0$, even in the exceptional world.

In this chapter, we develop an extension over RETT that allows enforcing purity over terms: the `param` modality. To support this modality, we require some structures that are generated through a new syntactic translation. This new translation is used along the translation that interprets RETT. We then extend the source theory with the modality that enforces when a term must be pure and finally, we explain the relation between the extension and the new syntactic translation.

4.2 Reasonably Parametric Exceptional Translation

Currently, only propositional terms are pure and it is not possible to establish if computational terms are pure. It is possible to enforce this using parametricity techniques. The same idea was applied on [18] (see Section 2.5.3). The problem in that translation is that parametricity is applied indiscriminately on every term. This led to the problems already presented where it is not possible to reason about exceptions. One approach is to tweak the translation to work over the reasonably exceptional translation and be parametric only over computational terms. However, we think that this is a strong requirement. Therefore, we opt for a less restrictive approach to the use of parametricity that adds more freedom to the translation and more control to the user. The translation will only be applied over inductive blocks, because these terms are the ones where we want to enforce purity.

The idea for the reasonably parametric exceptional translation is to enforce purity only over one kind of terms along a single translation. Here, kind means type. This means that along a context, the translation is also parametrized by an identifier, which is the type over which parametricity is applied.

Before we can define the reasonably parametric exceptional translation, we need an auxiliary function: `checkρψ(·)`. This function will be used on types and it extracts the relevant information to decide when parametricity should be applied to a given term. We can see its definition in Figure 4.1. In essence, it extracts the head (first argument) of the codomain of functions or the head of an application:

- `checkρψ(bool) ≡ bool`
- `checkρψ(vec nat 0) ≡ vec`
- `checkρψ(Π(A : □i)(a : A). T A a) ≡ T`

Note that even though the type system allows lambdas in types, the translation does not support it: it is undefined by `checkρψ(·)`. This means that we assume that the following kind of types, although valid, do not exist at the type level (lambda application at type level):

$$(\lambda(A : \square). A) \square \rightarrow \square$$

$$\begin{array}{l}
T[s]_{\rho\psi}^{\Gamma} := \lambda(A : [s]_{\psi}^{\Gamma}). [A]_{\psi}^{\Gamma} \rightarrow s \\
T[x]_{\rho\psi}^{\Gamma} := x_{\rho\psi} \\
T[\lambda(x : A). M]_{\rho\psi}^{\Gamma} := \begin{cases} \lambda(x : [A]_{\psi}^{\Gamma}) (x_{\rho\psi} : T[A]_{\rho\psi}^{\Gamma} x). T[M]_{\rho\psi}^{\Gamma, x:A} & \text{check}_{\rho\psi}(A) \equiv T \\ \lambda(x : [A]_{\psi}^{\Gamma}). T[M]_{\rho\psi}^{\Gamma, x:A} & \sim \end{cases} \\
T[M N]_{\rho\psi}^{\Gamma} := \begin{cases} T[M]_{\rho\psi}^{\Gamma} [N]_{\psi}^{\Gamma} T[N]_{\rho\psi}^{\Gamma} & \Gamma \vdash M : \Pi(x : A). B \wedge \text{check}_{\rho\psi}(A) \equiv T \\ T[M]_{\rho\psi}^{\Gamma} [N]_{\psi}^{\Gamma} & \sim \end{cases} \\
T[\Pi(x : A). B]_{\rho\psi}^{\Gamma} := \begin{cases} \lambda(f : \Pi(x : [A]_{\psi}^{\Gamma}). [B]_{\psi}^{\Gamma, x:A}). & \text{check}_{\rho\psi}(A) \equiv T \\ \quad \Pi(x : [A]_{\psi}^{\Gamma}) (x_{\rho\psi} : T[A]_{\rho\psi}^{\Gamma} x). T[B]_{\rho\psi}^{\Gamma, x:A} (f x) & \\ \lambda(f : \Pi(x : [A]_{\psi}^{\Gamma}). [B]_{\psi}^{\Gamma, x:A}). & \sim \\ \quad \Pi(x : [A]_{\psi}^{\Gamma}). T[B]_{\rho\psi}^{\Gamma, x:A} (f x) & \end{cases} \\
T[A]_{\rho\psi}^{\Gamma} := T[A]_{\rho\psi}^{\Gamma} \\
\hline
T[\cdot]_{\rho\psi} := \cdot \\
T[\Gamma, x : A]_{\rho\psi} := \begin{cases} [\Gamma]_{\psi}, x : [A]_{\psi}^{\Gamma}, x_{\rho\psi} : T[A]_{\rho\psi}^{\Gamma} x & \text{check}_{\rho\psi}(A) \equiv T \\ [\Gamma]_{\psi}, x : [A]_{\psi}^{\Gamma} & \sim \end{cases}
\end{array}$$

Figure 4.2: Reasonably Parametric Exceptional Translation Definition

In practice, this restriction can be bypassed by an explicit rewrite. Note that the check function leaves inductive elimination out. Later, it will be clear why this is not necessary. For now, this simple function will serve our purpose.

With the previous function, we can define the reasonably parametric exceptional translation $T[\cdot]_{\rho\psi}^{\Gamma}$. The translation applies parametricity in conjunction with the exceptional translation (just like [18]), but only to the terms that are related to T . This relation is an expectation over the result of $\text{check}_{\rho\psi}(\cdot)$ related to the current term being translated: for lambdas and binders, the domain should check to T ; and for applications, the domain of the left term's type should check to T . We can see the definition in Figure 4.2.

We will now prove the soundness theorem of this translation. We first need some lemmas and a new definition.

Definition 16 (Context containment). *Given two context Γ and Δ . We say that Δ contains Γ ($\Gamma \sqsubseteq \Delta$) if:*

1. $\text{dom}(\Gamma) \subseteq \text{dom}(\Delta)$.
2. *The relative order of identifiers in Γ is preserved in Δ .*
3. $\forall x : A \in \Gamma \Rightarrow x : B \in \Delta \wedge A \equiv B$.

In simple terms, this relation between contexts means that given two context Γ and Δ where $\Gamma \sqsubseteq \Delta$ holds, then Δ contains at least the same identifiers of Γ , where the associated

type in one context is equivalent to the associated type in the other for each mutual identifier between the contexts. Moreover, Δ can contain more identifiers than Γ , but these identifiers do not interfere with the mutual identifiers between Γ and Δ . It is important to remember that identifiers are unique.

Lemma 8. *Given a context Γ , and terms M and A . If $\Gamma \vdash M : A$, then $\vdash \Gamma$.*

Proof. By induction in the typing derivation. For each case, it is a direct consequence of the assumptions of the induction or a direct application of the induction hypothesis with WF-CONS. \square

Lemma 9. *Given contexts Γ and Δ , and a term A . If*

- $\Gamma, x : A \vdash x : A$
- $\vdash \Gamma, x : A, \Delta$

then $\Gamma, x : A, \Delta \vdash x : A$.

Proof. By induction on the form of Δ :

Case $\Delta' := \cdot$:

We need to prove $\Gamma, x : A \vdash x : A$, which is our assumption.

Case $\Delta := \Delta, y : B$:

We need to prove

$$\Gamma, x : A, \Delta', y : B \vdash x : A$$

We have:

1. $\vdash \Gamma, x : A, \Delta', y : B$ because we are doing induction over d .
2. $\Gamma, x : A, \Delta' \vdash B : s$ for some s by analysis on the structure of (1).
3. $\vdash \Gamma, x : A, \Delta'$ by Lemma 8 and (2).
4. $\Gamma, x : A, \Delta' \vdash x : A$ by the induction hypothesis along (3).

We can conclude by WEAKENING with (2) and (4). \square

Lemma 10. *Given a context Γ and a term A . If $x : A \in \Gamma$ and $\vdash \Gamma$, then we have $\Gamma \vdash x : A$.*

Proof. This is a corollary of Lemma 9 because if $x : A \in \Gamma$, then there exists Γ_1 and Γ_2 such that $\Gamma := \Gamma_1, x : A, \Gamma_2$ \square

Lemma 11. *Given two contexts Γ and Δ , and terms M and A . If we have $\Gamma \sqsubseteq \Delta$, $\vdash \Delta$ and $\Gamma \vdash M : A$, then we have $\Delta \vdash M : A$.*

Proof. By induction over the typing derivation:

Case TYPE, PROP:

It is direct because we have $\vdash \Delta$.

Case WEAKENING:

We need to prove

$$\Delta \vdash M : B$$

We have as induction hypothesis $\Gamma, x : A \sqsubseteq \Delta$, which means that $\Gamma \sqsubseteq \Delta$. With this hypothesis and using the induction hypothesis, we can conclude $\Delta \vdash M : B$.

Case TYPE-PROD, PROP-PROD, IMPRED:

These cases can be analyzed together assuming s_1, s_2, s_3 with the respective sort for each rule. We need to prove

$$\Delta \vdash \Pi(x : A). B : s_3$$

We have:

1. $\Gamma \sqsubseteq \Delta$
2. $\vdash \Delta$

We can derive:

3. $\Delta \vdash A : s_1$ by induction hypothesis.
4. $\Delta, x : A \vdash B : s_2$ by induction hypothesis and that $\Gamma \sqsubseteq \Delta \Rightarrow \Gamma, x : A \sqsubseteq \Delta, x : A$.

We can conclude by the corresponding product rule.

Case CONV:

We have to prove

$$\Delta \vdash M : A$$

We have by induction hypothesis

1. $\Delta \vdash M : B$
2. $\Delta \vdash A : s$

For conversion, the context is not used and it can be swapped.

3. $\Delta \vdash M \equiv N$

We conclude by applying CONV.

Case ABS:

We have to prove

$$\Delta \vdash \lambda(x : A). M : \Pi(x : A). B$$

In this case, we have:

1. $\Gamma \sqsubseteq \Delta$
2. $\vdash \Delta$

We can derive:

3. $\Delta \vdash \Pi(x : A). B : s$ by induction hypothesis.

4. $\Delta, x : A \vdash M : B$ by induction hypothesis and that $\Gamma \sqsubseteq \Delta \Rightarrow \Gamma, x : A \sqsubseteq \Delta, x : A$.

We conclude by ABS.

Case APP:

We have to prove

$$\Delta \vdash M N : B\{x := N\}$$

We have by induction hypothesis

1. $\Delta \vdash M : \Pi(x : A). B$
2. $\Delta \vdash N : A$

We conclude by APP.

Case IDEM:

We have to prove

$$\Delta \vdash x : A$$

We have $\Gamma, x : A \sqsubseteq \Delta$ by induction hypothesis. This implies that $x : A \in \Delta$. We conclude this case by Lemma 10

□

Lemma 12. *Given a context Γ , and terms M and A . Given that $[\Gamma]_\psi \vdash M : A$, then we have ${}^T[\Gamma]_{\rho\psi} \vdash M : A$.*

Proof. This is a corollary of Lemma 11 because $[\Gamma]_\psi \sqsubseteq {}^T[\Gamma]_{\rho\psi}$ by translation definition. □

Lemma 13. *Given a context Γ , and terms M , A and B . Given that $\Gamma \vdash {}^T[M]_{\rho\psi}^\Gamma : A$, then we have $\Gamma \vdash {}^T[M]_{\rho\psi}^{\Gamma, x : B} : A$ if $x \notin \text{FreeVar}(M)$.*

Proof. The reasonably parametric exceptional translation uses the context Γ to type M during the translation. If the context is extended with variables that are not free in M and identifiers are unique, then we have $\Gamma \sqsubseteq \Gamma, x : A$. By Lemma 11, we can swap the context and the translation will remain unaffected. This allow us to conclude. □

Lemma 14 (Substitution). *Given a context Γ and terms M , N , A and T , such that $\Gamma \vdash N : A$. Then, we have:*

$${}^T[M\{x := N\}]_{\rho\psi}^\Gamma \equiv {}^T[M]_{\rho\psi}^{\Gamma, x : A} \{x := [N]_\psi^\Gamma, x_{\rho\psi} := {}^T[N]_{\rho\psi}^\Gamma\}$$

Proof. If $\text{check}_{\rho\psi}(A) \not\equiv T$, then this substitution behaves just like Lemma 3 because the identifier $x_{\rho\psi}$ is unused along ${}^T[M]_{\rho\psi}^\Gamma$. In the other case, we assume $\text{check}_{\rho\psi}(A) \equiv T$. The proof is done by induction on M with a generalized context. From now on, the identifier y does not match the identifier being translated.

Case $M := \square_i$, $M := *$:

Holds trivially.

Case $M := x$, $M := y$:

Holds trivially.

Case $M := \lambda(y : A_y). M_1$:

We can assume that x is not the same as y , $y \notin \text{FreeVar}(N)$ and $y \notin \text{FreeVar}(A_y)$ by α -rename. Then, we get

$$T[\lambda(y : A_y\{x := N\}). M_1\{x := N\}]_{\rho\psi}^\Gamma$$

We do a case analysis over $\text{check}_{\rho\psi}(A_y)$. However, the case for $\text{check}_{\rho\psi}(A_y) \not\equiv T$ is contained in the case $\text{check}_{\rho\psi}(A_y) \equiv T$, thus we analyze the last. We have

$$\lambda(y : \llbracket A_y\{x := N\} \rrbracket_\psi^\Gamma) (y_{\rho\psi} : T \llbracket A_y\{x := N\} \rrbracket_{\rho\psi}^\Gamma y). T[M_1\{x := N\}]_{\rho\psi}^{\Gamma, y: A_y}$$

By applying the induction hypothesis and Lemma 3 we get

$$\lambda(y : \llbracket A_y \rrbracket_\psi^{\Gamma, x:A} \{x := \llbracket N \rrbracket_\psi^\Gamma\}) (y_{\rho\psi} : T \llbracket A_y \rrbracket_{\rho\psi}^{\Gamma, x:A} \{x := \llbracket N \rrbracket_{\rho\psi}^\Gamma, x_{\rho\psi} := T \llbracket N \rrbracket_{\rho\psi}^\Gamma\} y). \\ T[M_1]_{\rho\psi}^{\Gamma, y: A_y, x:A} \{x := \llbracket N \rrbracket_\psi^{\Gamma, y: A_y}, x_{\rho\psi} := T \llbracket N \rrbracket_{\rho\psi}^{\Gamma, y: A_y}\}$$

We also have:

$$T[M_1]_{\rho\psi}^{\Gamma, y: A_y, x:A} \equiv T[M_1]_{\rho\psi}^{\Gamma, x:A, y: A_y}$$

Because x does not affect typing of A_y , and y is not used for typing A . Thus, it is possible to fold the substitution and translation, which gives the result.

Case $M := \Pi(x : A). B$:

This case has some similarities with the previous one. Note that binders are wrapped over abstractions, for which we already proved that it holds. It remains to show how binders interact with substitution, which can be easily derived from the abstraction case, replacing abstractions by binders.

Case $M := M_1 M_2$:

Straightforward application of induction hypothesis over subterms M_1 and M_2

□

Lemma 15 (Conversion). *Given a context Γ and terms M, N and T , such that $\Gamma \vdash M \equiv N$, then $\Gamma \vdash T[M]_{\rho\psi}^\Gamma \equiv T[N]_{\rho\psi}^\Gamma$*

Proof. By induction on the conversion

Case β -reduction:

We have the following $M := (\lambda(x : A). M_1) N_1$ and $N := M_1\{x := N_1\}$. We perform a case analysis on the result of $\text{check}_{\rho\psi}(A)$

Subcase $\text{check}_{\rho\psi}(A) \equiv T$:

We have the following:

- $T[M]_{\rho\psi}^\Gamma := (\lambda(x : \llbracket A \rrbracket_\psi^\Gamma) (x_{\rho\psi} : T \llbracket A \rrbracket_{\rho\psi}^\Gamma x). T[M_1]_{\rho\psi}^{\Gamma, x:A}) \llbracket N_1 \rrbracket_\psi^\Gamma T \llbracket N_1 \rrbracket_{\rho\psi}^\Gamma$ by definition of the translation
- $T[M_1]_{\rho\psi}^{\Gamma, x:A} \{x := \llbracket N_1 \rrbracket_\psi^\Gamma, x_{\rho\psi} := T \llbracket N_1 \rrbracket_{\rho\psi}^\Gamma\}$ by Lemma 14

This holds by a double application of β -reduction.

Subcase $\text{check}_{\rho\psi}(A) \not\equiv T$:

Similar to the previous case.

Case η -expansion:

We have to take into account that abstractions are translated as abstractions (simple or double depending on the type of the binder). Through case analysis similar to the previous one, we conclude that η -expansion is respected.

Case application:

Straightforward application of induction hypothesis and conversion rules. □

Theorem 16 (Reasonably Parametric Exceptional Translation Soundness). *Given a context Γ , terms M , A and T , such that $\Gamma \vdash M : A$ and $\text{check}_{\rho\psi}(A) \equiv T$, then:*

$${}^T \llbracket \Gamma \rrbracket_{\rho\psi} \vdash {}^T \llbracket M \rrbracket_{\rho\psi}^\Gamma : {}^T \llbracket A \rrbracket_{\rho\psi}^\Gamma \llbracket M \rrbracket_\psi^\Gamma$$

Proof. By induction on the typing derivation

Case TYPE, PROP:

We start from $\Gamma \vdash s_1 : s_2$ and need to prove ${}^T \llbracket \Gamma \rrbracket_{\rho\psi} \vdash \lambda(A : \llbracket s_1 \rrbracket_\psi^\Gamma). \llbracket A \rrbracket_\psi^\Gamma \rightarrow s_1 : \llbracket s_1 \rrbracket_\psi^\Gamma \rightarrow s_2$. We can conclude by applying ABS if we can derive:

1. ${}^T \llbracket \Gamma \rrbracket_{\rho\psi} \vdash \llbracket s_1 \rrbracket_\psi^\Gamma \rightarrow s_2 : s_3$ for some s_3
2. ${}^T \llbracket \Gamma \rrbracket_{\rho\psi}, A : \llbracket s_1 \rrbracket_\psi^\Gamma \vdash \llbracket A \rrbracket_\psi^\Gamma \rightarrow s_1 : s_2$

For (1), we have:

3. $\llbracket \Gamma \rrbracket_\psi \vdash \llbracket s_1 \rrbracket_\psi^\Gamma : \llbracket s_2 \rrbracket_\psi^\Gamma$ by Theorem 5
4. ${}^T \llbracket \Gamma \rrbracket_{\rho\psi} \vdash \llbracket s_1 \rrbracket_\psi^\Gamma : s_2$ derived from (3) applying Lemma 2 and Lemma 12

We can apply TYPE-PROD to conclude.

For (2), we have can derive:

5. ${}^T \llbracket \Gamma \rrbracket_{\rho\psi}, A : \llbracket s_1 \rrbracket_\psi^\Gamma \vdash \llbracket A \rrbracket_\psi^\Gamma : \llbracket s_1 \rrbracket_\psi^\Gamma$ because $\llbracket A \rrbracket_\psi^\Gamma \equiv A$
6. ${}^T \llbracket \Gamma \rrbracket_{\rho\psi}, A : \llbracket s_1 \rrbracket_\psi^\Gamma \vdash \llbracket A \rrbracket_\psi^\Gamma : s_1$ by Lemma 2 over (5)

With the previous hypotheses, we can conclude the second case by TYPE-PROD or PROP-PROD depending the case.

Case TYPE-PROD, PROP-PROD, IMPRED:

For these cases we represent the sorts involved in the typing rules as s_1 , s_2 and s_3 with the expected mapping. We need to prove

$${}^T \llbracket \Gamma \rrbracket_{\rho\psi} \vdash \lambda(f : \Pi(x : \llbracket A \rrbracket_\psi^\Gamma). \llbracket B \rrbracket_\psi^{\Gamma, x:A}). F : (\Pi(x : \llbracket A \rrbracket_\psi^\Gamma). \llbracket B \rrbracket_\psi^{\Gamma, x:A}) \rightarrow s_3$$

where F could be two possible values:

- $\Pi(x : \llbracket A \rrbracket_\psi^\Gamma). {}^T \llbracket B \rrbracket_{\rho\psi}^{\Gamma, x:A} (f x)$
- $\Pi(x : \llbracket A \rrbracket_\psi^\Gamma) (x_{\rho\psi} : {}^T \llbracket A \rrbracket_{\rho\psi}^\Gamma x). {}^T \llbracket B \rrbracket_{\rho\psi}^{\Gamma, x:A} (f x)$

We need to derive the following in order to conclude:

- (1) ${}^T \llbracket \Gamma \rrbracket_{\rho\psi} \vdash (\Pi(x : \llbracket A \rrbracket_\psi^\Gamma). \llbracket B \rrbracket_\psi^{\Gamma, x:A}) \rightarrow s_3 : s_4$ for some s_4

$$(2) \quad T \llbracket \Gamma \rrbracket_{\rho\psi}, f : \Pi(x : \llbracket A \rrbracket_{\psi}^{\Gamma}). \llbracket B \rrbracket_{\psi}^{\Gamma, x:A} \vdash F : s_3$$

We have the following as induction hypothesis:

$$(3) \quad \Gamma \vdash A : s_1$$

$$(4) \quad \Gamma, x : A \vdash B : s_2$$

$$(5) \quad \text{Can derive } \Gamma \vdash \Pi(x : A). B : s_3$$

To derive the first judgment, we take hypothesis (5) and apply Theorem 5, Lemma 2, Lemma 12 and Lemma 1 to get $T \llbracket \Gamma \rrbracket_{\rho\psi} \vdash \Pi(x : \llbracket A \rrbracket_{\psi}^{\Gamma}). \llbracket B \rrbracket_{\psi}^{\Gamma, x:A} : s_3$. The rest of (1) is direct for the corresponding s_4 .

To derive (2) we need to prove:

$$(6) \quad T \llbracket \Gamma \rrbracket_{\rho\psi}, f : \Pi(x : \llbracket A \rrbracket_{\psi}^{\Gamma}). \llbracket B \rrbracket_{\psi}^{\Gamma, x:A} \vdash \llbracket A \rrbracket_{\psi}^{\Gamma} : s_1$$

$$(7) \quad T \llbracket \Gamma \rrbracket_{\rho\psi}, f : \Pi(x : \llbracket A \rrbracket_{\psi}^{\Gamma}). \llbracket B \rrbracket_{\psi}^{\Gamma, x:A}, x : \llbracket A \rrbracket_{\psi}^{\Gamma} \vdash F' : s_2$$

where F' could be:

- $T \llbracket B \rrbracket_{\rho\psi}^{\Gamma, x:A} (f x)$
- $\Pi(x_{\rho\psi} : T \llbracket A \rrbracket_{\rho\psi}^{\Gamma} x). T \llbracket B \rrbracket_{\rho\psi}^{\Gamma, x:A} (f x)$

Judgment (6) can be derived from (3) and applying Theorem 5, Lemma 2, Lemma 12, extending context property of CIC and unique identifiers. For judgment (7), we can assume, without loss of generality, that $F' := \Pi(x_{\rho\psi} : T \llbracket A \rrbracket_{\rho\psi}^{\Gamma} x). T \llbracket B \rrbracket_{\rho\psi}^{\Gamma, x:A} (f x)$ which implies that $\text{check}_{\rho\psi}(A) \equiv T$ (the other case is *similar*). We need to prove:

$$(8) \quad T \llbracket \Gamma \rrbracket_{\rho\psi}, f : \Pi(x : \llbracket A \rrbracket_{\psi}^{\Gamma}). \llbracket B \rrbracket_{\psi}^{\Gamma, x:A}, x : \llbracket A \rrbracket_{\psi}^{\Gamma} \vdash T \llbracket A \rrbracket_{\rho\psi}^{\Gamma} x : s_1$$

$$(9) \quad T \llbracket \Gamma \rrbracket_{\rho\psi}, f : \Pi(x : \llbracket A \rrbracket_{\psi}^{\Gamma}). \llbracket B \rrbracket_{\psi}^{\Gamma, x:A}, x : \llbracket A \rrbracket_{\psi}^{\Gamma}, x_{\rho\psi} : T \llbracket A \rrbracket_{\rho\psi}^{\Gamma} x \vdash T \llbracket B \rrbracket_{\rho\psi}^{\Gamma, x:A} (f x) : s_1$$

For (8), it can be proved by the induction hypothesis over (3), which gives

$$T \llbracket \Gamma \rrbracket_{\rho\psi} \vdash T \llbracket A \rrbracket_{\rho\psi}^{\Gamma} : \llbracket A \rrbracket_{\psi}^{\Gamma} \rightarrow s_1 \equiv T \llbracket \Gamma \rrbracket_{\rho\psi} \vdash T \llbracket A \rrbracket_{\rho\psi}^{\Gamma} : \llbracket A \rrbracket_{\psi}^{\Gamma} \rightarrow s_1$$

We can prove it by extending context property and APP.

Finally, we need to derive the typing judgement (9). The induction hypothesis over (4) gives us (according to the value of F'):

$$\begin{aligned} & T \llbracket \Gamma \rrbracket_{\rho\psi}, x : \llbracket A \rrbracket_{\psi}^{\Gamma}, x_{\rho\psi} : T \llbracket A \rrbracket_{\rho\psi}^{\Gamma} x \vdash T \llbracket B \rrbracket_{\rho\psi}^{\Gamma} : \llbracket B \rrbracket_{\psi}^{\Gamma} \rightarrow s_1 \\ & \quad \equiv \\ & T \llbracket \Gamma \rrbracket_{\rho\psi}, x : \llbracket A \rrbracket_{\psi}^{\Gamma}, x_{\rho\psi} : T \llbracket A \rrbracket_{\rho\psi}^{\Gamma} x \vdash T \llbracket B \rrbracket_{\rho\psi}^{\Gamma} : \llbracket B \rrbracket_{\psi}^{\Gamma} \rightarrow s_1 \end{aligned}$$

Given the context of (9), we can derive that the type of $(f x)$ is $\llbracket B \rrbracket_{\psi}^{\Gamma, x:A}$. Along the previous induction hypothesis, we can conclude the desired result by extending the context and APP.

Case WEAKENING:

By a case analysis over the result of $\text{check}_{\rho\psi}(A)$

Subcase $\text{check}_{\rho\psi}(A) \equiv T$:

We need to prove

$$T[\Gamma]_{\rho\psi}, x : [A]_{\psi}^{\Gamma}, x_{\rho\psi} : T[A]_{\rho\psi}^{\Gamma} \quad x \vdash T[M]_{\rho\psi}^{\Gamma, x:A} : T[B]_{\rho\psi}^{\Gamma} [M]_{\psi}^{\Gamma}$$

We have as hypothesis:

- (1) $\Gamma \vdash M : B$
- (2) $\Gamma \vdash A : s$

We can derive:

- (3) $T[\Gamma]_{\rho\psi} \vdash T[M]_{\rho\psi}^{\Gamma} : T[B]_{\rho\psi}^{\Gamma} [M]_{\psi}^{\Gamma}$ by induction hypothesis over (1)
- (4) $T[\Gamma]_{\rho\psi} \vdash [A]_{\psi}^{\Gamma} : s$ by Theorem 5, Lemma 2 and Lemma 12
- (5) $T[\Gamma]_{\rho\psi}, x : [A]_{\psi}^{\Gamma} \vdash T[A]_{\rho\psi}^{\Gamma} : [A]_{\psi}^{\Gamma} \rightarrow s$ by induction hypothesis over (2), translation definition and WEAKENING with (4)
- (6) $T[\Gamma]_{\rho\psi}, x : [A]_{\psi}^{\Gamma} \vdash T[A]_{\rho\psi}^{\Gamma} \quad x : s$ by APP and (5)

With all the previous derivations, we can conclude:

$$T[\Gamma]_{\rho\psi}, x : [A]_{\psi}^{\Gamma}, x_{\rho\psi} : T[A]_{\rho\psi}^{\Gamma} \quad x \vdash T[M]_{\rho\psi}^{\Gamma} : T[B]_{\rho\psi}^{\Gamma} [M]_{\psi}^{\Gamma}$$

using (3) and (6) along WEAKENING. Finally, we apply Lemma 13 because we can assume that $x \notin \text{FreeVar}(M)$, which concludes this case.

Subcase $\text{check}_{\rho\psi}(A) \not\equiv T$:

This case is similar to the previous.

Case ABS:

By a case analysis over the result of $\text{check}_{\rho\psi}(A)$ assuming $M := \lambda(x : A). N$

Subcase $\text{check}_{\rho\psi}(A) \equiv T$:

We need to prove

$$T[\Gamma]_{\rho\psi} \vdash \lambda(x : [A]_{\psi}^{\Gamma}) (x_{\rho\psi} : T[A]_{\rho\psi}^{\Gamma} \quad x). T[M]_{\rho\psi}^{\Gamma, x:A} : \Pi(x : [A]_{\psi}^{\Gamma}) (x_{\rho\psi} : T[A]_{\rho\psi}^{\Gamma} \quad x). T[B]_{\rho\psi}^{\Gamma, x:A} [M]_{\psi}^{\Gamma, x:A}$$

The type of the previous judgment is reduced. The initial type is:

$$\Pi(x : [A]_{\psi}^{\Gamma}) (x_{\rho\psi} : T[A]_{\rho\psi}^{\Gamma} \quad x). T[B]_{\rho\psi}^{\Gamma, x:A} ((\lambda(x : [A]_{\psi}^{\Gamma}). [M]_{\psi}^{\Gamma, x:A}) \quad x)$$

Note that $x \notin \text{FreeVar}(A)$ by hypothesis of the calculus. By applying β -reduction on the lambda we get

$$\Pi(x : [A]_{\psi}^{\Gamma}) (x_{\rho\psi} : T[A]_{\rho\psi}^{\Gamma} \quad x). T[B]_{\rho\psi}^{\Gamma, x:A} [M]_{\psi}^{\Gamma, x:A} \{x := x\}$$

Noting that $x \in \text{FreeVar}(M)$, using Lemma 3 and the fact that replacing by the same binder does not change anything, we get the expected type. In order to conclude, we use an iteration of ABS. We need to derive judgements related to the type of the lambda which is similar to what we have done for the binders cases. The only derivation that is still missing is:

$$\Gamma, x : [A]_{\psi}^{\Gamma}, x_{\rho\psi} : T[A]_{\rho\psi}^{\Gamma} \quad x \vdash T[M]_{\rho\psi}^{\Gamma, x:A} : T[B]_{\rho\psi}^{\Gamma, x:A} [M]_{\psi}^{\Gamma, x:A}$$

This can be concluded by the induction hypothesis.

Subcase $\text{check}_{\rho\psi}(A) \not\equiv T$:

This case is similar to the previous one without the extra binder in the context.

Case APP:

Here, it is necessary to perform a case analysis over $\text{check}_{\rho\psi}(A)$

Subcase $\text{check}_{\rho\psi}(A) \equiv T$:

We need to prove:

$$T \llbracket \Gamma \rrbracket_{\rho\psi} \vdash T \llbracket M \rrbracket_{\rho\psi}^\Gamma \llbracket N \rrbracket_{\psi}^\Gamma \quad T \llbracket N \rrbracket_{\rho\psi}^\Gamma : T \llbracket B\{x := N\} \rrbracket_{\rho\psi}^\Gamma (\llbracket M \rrbracket_{\psi}^\Gamma \llbracket N \rrbracket_{\psi}^\Gamma)$$

Note that here we can apply Lemma 14 to get

$$T \llbracket B\{x := N\} \rrbracket_{\rho\psi}^\Gamma (\llbracket M \rrbracket_{\psi}^\Gamma \llbracket N \rrbracket_{\psi}^\Gamma) \equiv T \llbracket B \rrbracket_{\rho\psi}^{\Gamma, x:A} \{x := \llbracket N \rrbracket_{\psi}^\Gamma, x_{\rho\psi} := T \llbracket N \rrbracket_{\rho\psi}^\Gamma\} (\llbracket M \rrbracket_{\psi}^\Gamma \llbracket N \rrbracket_{\psi}^\Gamma)$$

We get as induction hypotheses:

1. $T \llbracket \Gamma \rrbracket_{\rho\psi} \vdash T \llbracket M \rrbracket_{\rho\psi}^\Gamma : (\lambda(f : F'). \Pi(x : \llbracket A \rrbracket_{\psi}^\Gamma) (x_{\rho\psi} : T \llbracket A \rrbracket_{\rho\psi}^\Gamma x). T \llbracket B \rrbracket_{\rho\psi}^{\Gamma, x:A} (f x)) \llbracket M \rrbracket_{\psi}^\Gamma$
2. $T \llbracket \Gamma \rrbracket_{\rho\psi} \vdash T \llbracket N \rrbracket_{\rho\psi}^\Gamma : T \llbracket A \rrbracket_{\rho\psi}^\Gamma \llbracket N \rrbracket_{\psi}^\Gamma$

We can derive:

3. $T \llbracket \Gamma \rrbracket_{\rho\psi} \vdash T \llbracket M \rrbracket_{\rho\psi}^\Gamma : \Pi(x : \llbracket A \rrbracket_{\psi}^\Gamma) (x_{\rho\psi} : T \llbracket A \rrbracket_{\rho\psi}^\Gamma x). T \llbracket B \rrbracket_{\rho\psi}^{\Gamma, x:A} (\llbracket M \rrbracket_{\psi}^\Gamma x)$ by β -reduction on (1)
4. $T \llbracket \Gamma \rrbracket_{\rho\psi} \vdash \llbracket N \rrbracket_{\psi}^\Gamma : \llbracket A \rrbracket_{\psi}^\Gamma$ by Theorem 5 and Lemma 12

Finally, we can conclude by a double application of APP with hypotheses (3), (4) and (2)

Subcase $\text{check}_{\rho\psi}(A) \not\equiv T$:

This case is similar to the previous one without (2).

Case CONV:

This case follows by the induction hypothesis on the premises and Lemma 15.

Case IDEM:

We need to prove

$$T \llbracket \Gamma, x : A \rrbracket_{\rho\psi} \vdash x_{\rho\psi} : T \llbracket A \rrbracket_{\rho\psi}^{\Gamma, x:A} x$$

Note that we have $\text{check}_{\rho\psi}(A) \equiv T$ because otherwise, the hypotheses of the theorem do not hold. We have as induction hypothesis:

1. $T \llbracket \Gamma \rrbracket_{\rho\psi} \vdash T \llbracket A \rrbracket_{\rho\psi}^\Gamma : \llbracket A \rrbracket_{\psi}^\Gamma \rightarrow s$

We can derive

2. $T \llbracket \Gamma \rrbracket_{\rho\psi} \vdash \llbracket A \rrbracket_{\psi}^\Gamma : s$ by Theorem 5, Lemma 2 and Lemma 12
3. $T \llbracket \Gamma \rrbracket_{\rho\psi}, x : \llbracket A \rrbracket_{\psi}^\Gamma \vdash T \llbracket A \rrbracket_{\rho\psi}^{\Gamma, x:A} : \llbracket A \rrbracket_{\psi}^\Gamma \rightarrow s$ by WEAKENING on (1) and using Lemma 13
4. $T \llbracket \Gamma \rrbracket_{\rho\psi}, x : \llbracket A \rrbracket_{\psi}^\Gamma \vdash x : \llbracket A \rrbracket_{\psi}^\Gamma$ by IDEM with (2)
5. $T \llbracket \Gamma \rrbracket_{\rho\psi}, x : \llbracket A \rrbracket_{\psi}^\Gamma \vdash T \llbracket A \rrbracket_{\rho\psi}^{\Gamma, x:A} x : s$ by APP with (3) and (4)

We conclude with IDEM and (5).

Case WF-EMPTY:

Trivial

Case WF-CONS:

This case is quite similar to the IDEM case, except that we use well foundedness instead of typing.

□

4.3 Parametric Modality

In the previous section, we defined a translation that uses parametricity to enforce purity over a particular type. In this section, we explain how can we use the previous translation to control exceptions. To achieve this, it is necessary to extend RETT at the source level and the target level. We now describe these extensions and in the next section, we define how these extensions are related.

We first analyze the target theory to see what the limitations are, and then review the source theory.

4.3.1 Extending the Target Theory

Currently, RETT is not able to capture the purity of a term. Remember that RETT is generated by the reasonably exceptional translation. Here, we explain how we can use the reasonably parametric exceptional translation along the reasonably exceptional translation to generate an extension on the target theory of RETT where it is possible to establish that a term must be pure. This means that the reasonably parametric exceptional translation does not generate a new type theory.

The intuition for extending RETT is that the parametric translation enforce purity over a single type during its translation. Also, *base* types in CIC are defined by inductive blocks. This means that we could enforce purity at the inductive level and it will cover all the interesting cases of the calculus. Finally, propositions do not require the enforcement because they are already pure. With this in mind, we redefine how inductive blocks are translated.

Definition 17 (Inductive Reasonably Exceptional Translation). *Given an inductive block $\mathbf{Ind}_n \{ \Delta_I := \Delta_C \}$ along the blocks defined in Definition 14, we define:*

$$\begin{aligned} \bullet \Delta_{I\rho\psi} &:= \left\{ I_{\rho\psi} : \Pi \left[\overrightarrow{p : P} \right]_{\psi}^{\Gamma} \cdot \Pi \left[\overrightarrow{u : U} \right]_{\psi}^{\Gamma, \vec{p} : \vec{P}} \cdot I_{\psi} \vec{p} \vec{u} \rightarrow * \mid (I : D) \in \Delta_I \text{ and } I \text{ arity of } \square \right\} \\ \bullet \Delta_{C\rho\psi} &:= \left\{ c_{\rho\psi} : I_c \llbracket C \rrbracket_{\rho\psi}^{\Gamma} [c]_{\psi}^{\Gamma} \mid (c : C) \in \Delta_C \text{ and } I_c \text{ arity of } \square \right\} \end{aligned}$$

setting locally:

$$I \llbracket I \rrbracket_{\rho\psi}^{\Gamma} := I_{\rho\psi}, \forall I \in \text{dom}(\Delta_I)$$

We redefine the reasonably exceptional translation of inductive blocks to:

$$\llbracket \mathbf{Ind}_n \{ \Delta_I := \Delta_C \} \rrbracket_{\psi}^{\Gamma} := \mathbf{Ind}_n \{ \Delta_{I\psi} := \Delta_{C\psi} \} \cup \mathbf{Ind}_n \{ \Delta_{I\rho\psi} := \Delta_{C\rho\psi} \}$$

The new generated parametric inductive block transforms each inductive into a predicate of purity for the related exceptional inductive (note that they are an arity of *). The exceptional constructor is not generated, thus it is excluded as a pure term, which is the desired goal. Note

$$\begin{array}{l}
\mathbf{Ind}_0 \left\{ \text{empty}_{\rho\psi} : \text{empty}_{\psi} \rightarrow * := \right\} \\
\mathbf{Ind}_0 \left\{ \begin{array}{l} \text{nat}_{\rho\psi} : \text{nat}_{\psi} \rightarrow * := \text{O}_{\rho\psi} : \text{nat}_{\rho\psi} \text{O}_{\psi} \\ \text{S}_{\rho\psi} : \Pi(n : \text{nat}_{\psi}). \text{nat}_{\rho\psi} n \rightarrow \text{nat}_{\rho\psi} (\text{S}_{\psi} n) \end{array} \right\} \\
\mathbf{Ind}_1 \left\{ \begin{array}{l} \text{nil}_{\rho\psi} : \Pi(A : \text{type}_i). \text{list}_{\rho\psi} A (\text{nil}_{\psi} A) \\ \text{list}_{\rho\psi} : \Pi(A : \text{type}_i). \text{list}_{\psi} \rightarrow * := \text{cons}_{\rho\psi} : \Pi(A : \text{type}_i). \Pi(a : \text{El}_i A) (l : \text{list}_{\psi} A). \\ \text{list}_{\rho\psi} A l \rightarrow \text{list}_{\rho\psi} A (\text{cons}_{\psi} A a l) \end{array} \right\}
\end{array}$$

Figure 4.3: Examples of inductive blocks generated by the parametric translation

that the above definition is ambiguous for the case that the blocks only define propositions. We assume that the translation drops the right side of the union in such case.

Lemma 17. *The redefinition described in Definition 17 is sound*

Proof. We must establish that WF-IND holds for the extension. We already established the soundness of the translation. Also, I_{ψ} exists $\forall I, I \in \Delta_I$ when using the parametric translation. Finally, the translation respects the positivity conditions. \square

A few things to note. The previous redefinition modifies the target theory of RETT but not its source. For each translated inductive, a parametric inductive is generated. This parametric inductive can enforce the purity of the exceptional inductive at the type level in the target theory. We can see examples of parametric inductives in Figure 4.3. As an example, the following type captures the essence of the identity function for nat 's but allowing only pure terms:

$$\Pi(n : \text{nat}_{\psi}). \text{nat}_{\rho\psi} n \rightarrow \text{nat}_{\psi}$$

This type describes the expected purity because the second argument of the given type can not be generated if the first one is an exception: it is not possible to create the predicative witness for $\text{nat}_{\rho\psi} (\text{nat}_{\phi} e)$.

Moreover, given an inductive type I , it is not possible to create a predicate witness for

$$I_{\rho\psi} \vec{p} \vec{i} (I_{\phi} \vec{p} \vec{i} e)$$

This is a meta property of RETT, but it is not expressible.

4.3.2 Extending the Source Theory: Adding the Modality

We extended the translation and it is possible to express when a term must be pure in the target theory. We would like to be able to express it in the source theory. Take the last example of the previous section:

$$\Pi(n : \text{nat}_{\psi}). \text{nat}_{\rho\psi} n \rightarrow \text{nat}_{\psi}$$

At the target level, we have to use the dedicated inductive type $\text{nat}_{\rho\psi}$ associated to nat in order to declare purity. At the source level, we would like to be agnostic over the dedicated type and let the translation figure it out:

$$\Pi(n : \text{nat}). \mathcal{P} n \rightarrow \text{nat}$$

Here, we are making explicit that we want a proof of purity for the second argument through $\mathcal{P}n$. We call \mathcal{P} the *modality*. In addition, we want only computational terms to be enforced to be pure, because propositional terms are already pure. We will define the extensions needed to support the modality in the rest of the section.

The first step is to extend the terms of the calculus:

$$A, B, N, M, R, S := \dots \mid \mathcal{P}N$$

The new term, $\mathcal{P}N$, represents that the term N should be pure. This new term alone is not enough. We need to add a new typing judgment in order to have a valid system:

$$\frac{\text{PARAM-TERM} \quad A \text{ inductive} \quad \Gamma \vdash A : \square_i \quad \Gamma \vdash N : A}{\Gamma \vdash \mathcal{P}N : *}$$

These extensions, albeit arbitrary, will be proved valid by the translation (due to [3]). They permit us to enforce purity in the source theory. The A in the previous rule correspond to an inductive type fully applied and the extensions are not applicable to, for example, function types ($\text{nat} \rightarrow \text{nat}$).

Equipped with these extensions, we are able to define properties over the terms of the language. However, these properties are not general: it is possible to generate them for each specific type, but not quantify over them, just like the catch induction.

The first property is a consistency proof. Semantically, a term of type $\mathcal{P}(\text{raise } A \ e)$, for some generic type that depends on the context, should not be a valid term, because **raise** is not pure. However, it is not possible to prove this quantifying over all types, but it is possible to generate the proof for each type specifically. In CIC, if something is *false*, than it is possible to derive **False**. This means that we can add the following rule and later be able to translate it

$$\frac{\text{PARAM-RAISE} \quad A \text{ inductive} \quad \Gamma \vdash A : \square_i}{\Gamma \vdash \mathfrak{F} : \Pi(e : \mathbf{E}). \mathcal{P}(\text{raise } A \ e) \rightarrow \text{False}}$$

Namely, there exists a consistency proof for all inductive types that are an arity of \square_i .

The second property is an induction principle. This induction principle is similar to the *basic* induction principle, but it is required the proof of purity. This means that it is not necessary to cover the case when such a type can be an exception. For example, the following term corresponds to the *parametric induction* principle for **nat**

$$\begin{aligned} & \Pi(P : \text{nat} \rightarrow *). \\ & P \ 0 \rightarrow \\ & (\Pi(n : A). P \ n \rightarrow P \ (\mathbb{S} \ n)) \rightarrow \\ & \Pi(n : \text{nat}). \mathcal{P} \ n \rightarrow P \ n \end{aligned}$$

Note that this induction principle can only be used in $*$, because behind the scenes, it uses induction over parametricity proof to determine that the exceptional branch is not required, which restricts the elimination sort to $*$.

4.4 Modality Realization

In this section we will be connecting the two extensions developed in the previous section, extending the translation and taking into consideration the new terms and proving the soundness of the new typing judgment. Intuitively, the base extensions and first property can be seen as adding operations over each particular type, while the second property corresponds to generate a new term with the specified type.

Adding operations over types in functional languages can be modeled by type classes (more information in [23]), which will be how the modality is interpreted in the target theory. The following type class lives in the target theory.

```

Class Paramψ (A : typei) :=
  paramψ : Eli A → *
  param_soundnessψ : Π(e : ℰ). paramψ (Erri A e) → Falseψ

```

This type class models the previous extensions. First, we will extend the translation to consider the modality.

Definition 18 (Modality extension). *We extend the reasonably exceptional translation with:*

$$[\mathcal{P}]_{\psi}^{\Gamma} := \text{param}_{\psi}$$

Second, we need to prove that the typing judgments hold in the target theory. To achieve this, it is important to highlight that the modality can only be applied over inductive types in the source theory, which suggests that we have to declare an instance of the type class for every translated inductive type. We will show the content of such instance assuming a generic inductive type I with parameters $\vec{p} : P$ and indexes $\vec{u} : U$.

The type of the instance will be:

$$\Pi[\vec{p} : P]_{\psi}^{\Gamma} [\vec{u} : U]_{\psi}^{\Gamma}. \text{Param}_{\psi} (\text{TypeVal}_i (I_{\psi} \vec{p} \vec{u}) (I_{\phi} \vec{p} \vec{u}))$$

The first field of the type class is straightforward, considering that

$$\text{El}_i (\text{TypeVal}_i (I_{\psi} \vec{p} \vec{u}) (I_{\phi} \vec{p} \vec{u})) \equiv I_{\psi} \vec{p} \vec{u}$$

we get:

$$\bullet \text{param}_{\psi_I} := \lambda(m : I_{\psi} \vec{p} \vec{u}). I_{\rho\psi} \vec{p} \vec{u} m$$

We used the parametric exceptional inductive and passed the exceptional inductive term to generate the purity predicate.

The second is more complicated. First, we can reduce the terms at the target theory as follows

$$\text{Err}_i (\text{TypeVal}_i (I_{\psi} \vec{p} \vec{u}) (I_{\phi} \vec{p} \vec{u})) e \equiv I_{\phi} \vec{p} \vec{u} e$$

which implies that

$$\text{param}_{\psi} (I_{\phi} \vec{p} \vec{u} e) \equiv I_{\rho\psi} \vec{p} \vec{u} (I_{\phi} \vec{p} \vec{u} e)$$

With the previous equations, the second field is

- $\text{param_soundness}_{\psi_I} := \lambda(e : \mathbb{E}) (f : I_{\rho\psi} \vec{p} \vec{u} (I_\phi \vec{p} \vec{u} e)). ?$

The construction of the body is not easy, as we have to generate equations over the parametric exceptional inductive to prove that no constructor can generate a term where the last term is an exception, but is doable in the current settings. We will assume that inversion_I is such term which allows us to define the second field as

- $\text{param_soundness}_{\psi_I} := \lambda(e : \mathbb{E}) (f : I_{\rho\psi} \vec{p} \vec{u} (I_\phi \vec{p} \vec{u} e)). \text{inversion}_I$

The previous terms allow us to declare the instance of the corresponding type class. It only remains to proof the soundness. We first modify the translation of inductives

Definition 19 (Inductive Reasonably Exceptional Translation). *Given an inductive block $\mathbf{Ind}_n \{ \Delta_I := \Delta_C \}$. Along the blocks defined in Definition 17, we define:*

$$\Delta_{\text{Inst}\psi} := \left\{ \begin{array}{l} \mathfrak{J}_I : \Pi \overrightarrow{[[p : P]]_\psi}^\Gamma \cdot \Pi \overrightarrow{[[u : U]]_\psi}^{\Gamma, p : \vec{P}} \\ \text{Param}_\psi (\text{TypeVal}_i (I_\psi \vec{p} \vec{u}) (I_\phi \vec{p} \vec{u})) \end{array} \middle| (I : D) \in \Delta_I \text{ and } I \text{ arity of } \square_i \right\}$$

Where each \mathfrak{J}_I correspond to an instance of Param_ψ conformed by param_{ψ_I} and $\text{param_soudness}_{\psi_I}$. We redefine the reasonably exceptional translation of inductive blocks to:

$$[\mathbf{Ind}_n \{ \Delta_I := \Delta_C \}]_\psi^\Gamma := \mathbf{Ind}_n \{ \Delta_{I\psi} := \Delta_{C\psi} \} \cup \mathbf{Ind}_n \{ \Delta_{I\rho\psi} := \Delta_{C\rho\psi} \} \cup \Delta_{\text{Inst}\psi}$$

Just like Definition 17, the above definition is ambiguous for the case where the blocks only define propositions. We assume for this case that the translation drops the two rightmost contexts of the union.

We are now able to define the soundness of the modality.

Theorem 18 (Modality Soundness). *The extension of Definition 19 is sound with respect to the modality typing judgments.*

Proof. We need to prove that the new instances are a valid declaration and that we can realize in the target theory the two typing judgments of the source theory.

Case Valid instances:

Type classes can be thought as a single constructor inductive type where each field is an argument of the constructor. It is clear from the construction that the instance generation corresponds to the expected types and the positivity condition is respected. This means that the translation redefinition of inductive types is valid.

Case PARAM-TERM:

We need to show that

$$\Gamma \vdash [\mathcal{P} N]_\psi^\Gamma : [*]_\psi^\Gamma$$

Unfolding the translation we get that we have to prove

$$\Gamma \vdash \text{param}_\psi N : *$$

The hypotheses allow us to conclude that there exists an instance for A of the type class Param_ψ . It is then direct that the required judgments holds.

Case PARAM-RAISE:

We need to prove that

$$\Gamma \vdash [\mathfrak{F}]_{\psi}^{\Gamma} : \Pi(e : [\mathbf{E}]_{\psi}^{\Gamma}). [\mathcal{P}(\text{raise } A \ e)]_{\psi}^{\Gamma, e, \mathbf{E}} \rightarrow [\mathbf{False}]_{\psi}^{\Gamma, e, \mathbf{E}}$$

Unfolding definitions and reducing terms, we have to prove

$$\Gamma \vdash \mathfrak{F}_{\psi} : \Pi(e : \mathbb{E}). \text{param}_{\psi}(\text{Err}_i \ A \ e) \rightarrow \text{False}_{\psi}$$

Similar to the previous case, we can conclude that there exists an instance for A of the type class Param_{ψ} . The required element can be extracted of the instance and thus proving the required judgment. □

We have demonstrated that the modality is a sound extension and can be used to enforce purity over exceptional terms on demand (not flooding the entire calculus).

Finally, we are able to generate the parametric inductive elimination described in the previous section, where \mathcal{P} is replaced by param . For example, we showed that for nat was:

$$\begin{aligned} & \Pi(P : \text{nat} \rightarrow *). \\ & P \ 0 \rightarrow \\ & (\Pi(n : A). P \ n \rightarrow P \ (\mathbf{S} \ n)) \rightarrow \\ & \Pi(n : \text{nat}). \mathcal{P} \ n \rightarrow P \ n \end{aligned}$$

In this new setting, we now have that:

$$\begin{aligned} & \Pi(P : \text{nat} \rightarrow *). \\ & P \ 0 \rightarrow \\ & (\Pi(n : A). P \ n \rightarrow P \ (\mathbf{S} \ n)) \rightarrow \\ & \Pi(n : \text{nat}). \text{param} \ n \rightarrow P \ n \end{aligned}$$

In this chapter, we extended RETT with a modality that allows establishing the purity of terms on demand. This offers more fine grained control on terms, which in turn allows enunciating more expressive theorems. This extension is also validated through a syntactic translation into CIC, but it relies on the type class mechanism. In the next chapter, we *instantiate* RETT into the proof assistant Coq.

Chapter 5

RETT Implementation: A Coq Plugin

So far, we developed a new type theory that provides exceptions, the ability to reason about them consistently, and a modality to enforce the purity of terms. We now describe an actual implementation of RETT as a Coq plugin (CoqRETT)¹ where the terms of RETT are interpreted as Coq terms/axioms and compiled to Coq terms through the plugin.

The rest of the section is dedicated to assessing how CoqRETT can be used to reason about exceptional programs, its limitations, and how CoqRETT is implemented, i.e., the interpretation of RETT in Coq.

5.1 CoqRETT in Action

We will see how CoqRETT can be used to reason about exceptional terms. Remember that every term in RETT is interpreted through the translation. In CoqRETT, this step must be declared in the source code.

First, CoqRETT is loaded by

```
Require Import Effects.Exception.
```

This line adds the exceptional type, the raising function, the modality, and provides the translation and definition command.

```
Check Exception. (* Type *)
Check raise. (* ∀ (A: Type), Exception → A. *)
Check param. (* ∀ (A: Type), ParamMod A → A → Prop. *)
(* Effect Translate ... *)
(* Effect Definition ... *)
```

The command `Effect Translate` expects a Coq identifier and it tries to apply the translation over that term. The command `Effect Definition` allows to declare a constant in CoqRETT, but it is required to provide its translated term. For pedagogical reasons, we assume the following degenerate raising function:

¹<https://github.com/hfehrmann/reasonably-exceptional-type-theory>

```
Check raise. (* ∀ (A: Type), A. *)
```

Additionally, the modality is represented by the term `param`. Despite its type, the modality is simply used as

```
param M
```

Now, the following inductive is the type of list.

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

In order to interpret `list` as an exceptional inductive, we apply the translation as follows:

```
Effect Translate list.
```

When CoqRETT translates a computational inductive with this command, it makes available all the induction principles that RETT provides:

- The induction principle that considers the exceptional case when the term is being eliminated into a computational type:

```
Check list_catch.  
(* ∀ (A: Type) (P: list A → Type),  
  P nil →  
  (∀ (a: A) (l: list A), P l → P (cons a l)) →  
  P (raise (list A)) →  
  ∀ (l: list A), P l. *)
```

- The induction principle that considers the exceptional case when the term is being eliminated into a proposition:

```
Check list_catch_prop.  
(* ∀ (A: Type) (P: list A → Prop),  
  P nil →  
  (∀ (a: A) (l: list A), P l → P (cons a l)) →  
  P (raise (list A)) →  
  ∀ (l: list A), P l. *)
```

- The induction principle that does not consider the exceptional case because the term is pure and the term is being eliminated into a proposition:

```
Check list_ind_param.  
(* ∀ (A: Type) (P: list A → Prop),  
  P nil →  
  (∀ (a: A) (l: list A), P l → P (cons a l)) →  
  ∀ (l: list A), param l → P l. *)
```

The user must translate the induction principle.

```
Effect Translate list_rect.
```

Note that the computational rules that are valid in RETT cannot be added into CoqRETT. However, we can derive them as propositional equations. These terms are not generated automatically.

```
Effect Definition list_catch_nil_eq:
  ∀ A P Pnil Pcons Praise,
    list_catch A P Pnil Pcons Praise (nil A) = Pnil.
```

Proof.

reflexivity.

Defined.

```
Effect Definition list_catch_cons_eq:
  ∀ A P Pnil Pcons Praise a l,
    list_catch A P Pnil Pcons Praise (cons a l)
    =
    Pcons a l (list_catch A P Pnil Pcons Praise l).
```

Proof.

reflexivity.

Defined.

```
Effect Definition list_catch_raise_eq:
  ∀ A P Pnil Pcons Praise,
    list_catch A P Pnil Pcons Praise (raise (list A)) = Praise.
```

Proof.

reflexivity.

Defined.

```
Effect Definition list_rect_raise_eq:
  ∀ A P Pnil Pcons,
    list_rect A P Pnil Pcons (raise (list A)) = raise (P (raise (list A))).
```

Proof.

reflexivity.

Defined.

All of the above lays the ground to prove theorems related to lists. From now on, we will assume that the command `Effect Translate ...` is applied to every definition and theorem, which means that they are valid in RETT. For example, we are able to establish that an empty list cannot be an exception.

```
Theorem nil_not_raise: ∀ A, nil A ≠ raise (list A).
```

Proof.

```
  intros A.
  assert(
    H: ∀ l,
      nil A = l →
      list_catch A (fun _ => Prop) True (fun _ _ => True) False l
  ).
  { intros l Heq. destruct Heq. rewrite list_catch_nil_eq. exact I. }
  intro Heq. specialize (H (raise _)) Heq.
  rewrite list_catch_raise_eq in H. exact H.
```

Defined.

Here we can see a standard technique in Coq where a hypothesis is generalized to a proof of equality (`assert`) and then specialized (`specialize`) with the corresponding term. The same technique can be applied to prove that a `cons` of a list is not an exception:

Theorem `cons_not_raise`: $\forall A a l, \text{cons } a l \neq \text{raise } (\text{list } A)$.

This theorem is still valid when the given list is a `cons` with an exceptional list. For example

`cons 1 (raise (list A))`

We can also define the following functions for lists

Definition `head` $\{A: \text{Type}\} (l: \text{list } A): A :=$
`list_rect A (fun _ \Rightarrow A) (raise A) (fun a _ \Rightarrow a) l.`

Definition `tail` $\{A: \text{Type}\} (l: \text{list } A): A :=$
`list_rect A (fun _ \Rightarrow A) (raise (list A)) (fun _ l _ \Rightarrow l) l.`

Definition `length` $\{A: \text{Type}\} (l: \text{list } A): A :=$
`list_rect A (fun _ \Rightarrow nat) 0 (fun _ _ n \Rightarrow S n) l.`

Now, we would like to establish some properties of these functions with respect to how they behave when an exceptional list is given. First, we can prove the expected definitional equations.

Theorem `head_raise`: $\forall A, \text{head } (\text{raise } (\text{list } A)) = \text{raise } A$.

Theorem `tail_raise`: $\forall A, \text{tail } (\text{raise } (\text{list } A)) = \text{raise } (\text{list } A)$.

Theorem `length_raise`: $\forall A, \text{length } (\text{raise } (\text{list } A)) = \text{raise } \text{nat}$.

Theorem `length_cons_raise`: $\forall A a, \text{length } (\text{cons } a (\text{raise } (\text{list } A))) = \text{S } (\text{raise } \text{nat})$.

A more interesting theorem is that the `tail` function does not raise an exception if the list argument is not empty. To capture the notion of non empty, we use the \leq type, which is an inductively defined proposition.

Before we can prove such property, we have to prove that no natural number is *lower* than an exception.

Theorem `raise_not_leq`: $\forall n, n \leq \text{raise } \text{nat} \rightarrow \text{False}$.

Now, we can show that

Theorem `non_empty_list_distinct_tail_raise`:
 $\forall A l, 0 < \text{length } l \rightarrow \text{tail } l \neq \text{raise } (\text{list } A)$.

Lastly, we would like to prove that the head of a non empty list is not an exception.

Theorem `non_empty_list_head_not_raise`:
 $\forall A l, 0 < \text{length } l \rightarrow \text{head } l \neq \text{raise } A$.

However, this is not the case. CoqRETT allows the following lists

Definition `exceptional_list1`: `list nat` :=
`cons (raise nat) (cons (raise nat) (raise (list A)))`.

Definition `exceptional_list2`: `list nat` :=
`cons (raise nat) (cons (raise nat) nil)`.

These lists are non empty. Both contain exceptions as elements and the first one is also an exceptional list. The existence of both lists imply that the theorem for the `head` function is not provable without more constraints. This is where the modality enters. The modality is able to filter the first type of lists, because it operates at sub terms of the same type. Nevertheless, the second list is a pure list but still contains exceptional terms.

Not all is lost. In CoqRETT, we also have more flexibility on how we declare purity on terms which allows us to define a deep parametricity proposition for lists.

Definition `list_param_deep` $\{A\}$ $\{\text{Param } A\}$ $(l: \text{list } A): \text{Prop} :=$
`catch_list A (fun _ => Prop) True (fun a _ p => p \wedge param a) (fun _ => False) l`.

This proposition captures that a list must be pure, but also that its elements must be pure. Neither of the previous lists are deep pure. Then, the theorem for `head` can be stated and proved

Theorem `non_empty_deep_param_list_head_not_raise`:
 $\forall A l, 0 < \text{length } l \rightarrow \text{list_param_deep } l \rightarrow \text{head } l \neq \text{raise } A$.

This theorem is straightforward, but we have to use the consistency proof captured in the modality for the head element.

5.2 Limitations

CoqRETT is able to interpret exceptions. Nevertheless, the definitional equations that are valid for inductive eliminators can only be proved propositional in the source theory by a translation to the target theory. This means that, in practice, the equations of reduction must be declared after the translation of the inductives and cannot be used by the kernel to *compute*.

In addition, Coq has cumulative universes, which means that it is possible to raise exceptions on `Prop`. This means that in order to establish truly valid theorems in CoqRETT, it is necessary to translate every proof.

5.3 Foundation Code

The current development is a redefinition of a previous work [18]. As such, there exists a code base that implements the translations defined in Section 2.5.2 and Section 2.5.3 as a plugin. However, it was missing the implementation that handles records. Being able to translate records is required because the modality is represented as a type class, which are just records with a resolution mechanism. The code was updated to handle records as described in the same work [18].

5.4 Plugin foundation

We will show the foundation structures that are required by the translation. In the actual implementation, all terms that belong to the target theory are parametrized by an exceptional type argument. However, the current presentation will not be explicit about it, but it will refer to this term as \mathbb{E} .

First, we define the terms in the target theory, which handles exceptional types.

```
Cumulative Inductive type: Type :=
| TypeVal:  $\forall$  (A: Type), ( $\mathbb{E} \rightarrow A$ )  $\rightarrow$  type
| TypeErr:  $\mathbb{E} \rightarrow$  type
```

```
Definition El (A : type): Type :=
match A with
| TypeVal A _  $\Rightarrow$  A
| TypeErr e  $\Rightarrow$  unit
end.
```

```
Definition Err (A : type):  $\mathbb{E} \rightarrow$  El A :=
match A return  $\mathbb{E} \rightarrow$  El A with
| TypeVal _ e  $\Rightarrow$  e
| TypeErr _  $\Rightarrow$  fun _  $\Rightarrow$  tt
end.
```

Then, we define the two axioms, which are the exceptional type and raising function.

```
Axiom E: Type.
```

```
Axiom raise:  $\forall$  (e: E) (A: Type), A.
```

These are the terms that we use in CoqRETT at the source level. Now, every term must have its translated counterpart. The corresponding terms are

```
Definition E $_{\epsilon}$  := TypeVal  $\mathbb{E}$  (fun e  $\Rightarrow$  E).
```

```
Definition raise $_{\epsilon}$  (A: type) (e: E):= Err A e.
```

Finally, the modality is interpreted as a type class in CoqRETT. At the source level, the modality is defined as:

```
Class Param (A: Type): Type := {
  param: A  $\rightarrow$  Prop;
  param_correct:  $\forall$  (e: E), param (raise A e)  $\rightarrow$  False;
}
```

Here, we can see the modality `param`, but also, the consistency proof related to the type that declares the instance of the type class. Note that the consistency proof uses the `False` proposition. This forces the plugin to provide the translation of `False`.

```
Inductive False $_{\epsilon}$ : Prop :=.
```

The target counterpart of the modality is

```

Class Paramε (A: type): Type := {
  paramε: El A → Prop;
  param_correctε: ∀ (e: ℰ), paramε (raiseε A e) → Falseε;
}

```

5.5 Plugin Implementation

The plugin implements the translation described in the previous chapters. It has to take in consideration the structure of Coq. The plugin actually translate terms that are already defined in the environment – it translate the associated identifier. When the plugin translates an identifier, it translates the body and the type, generates a new identifier, associates the previous information and adds an entry to the environment. It then adds the identifiers into an internal map that relates the source and target identifiers. The plugin generates an initial map where the axioms and exceptional definitions described in the previous section are mapped, respectively.

On the negative fragment, the implementation is quite similar to the ETT plugin. However, as it was laid out in the previous chapters, we need to make a distinction when the current term is a proposition or a computational term. Both translations must carry a context to generate the expected universe constraints, but now it also must be used to check the current hierarchy of the term. Let us have the following definitions:

```

Definition type_id (A: Type) (a: A): A := a.
Check type_id. (* ∀ (A: Type), A → A. *)

Definition prop_id (A: Prop) (a: A): A := a.
Check prop_id. (* ∀ (A: Prop), A → A. *)

```

The first term is the identity on `Type` and the second term is the identity on `Prop`. Their respective translations are:

```

Definition type_idε (ℰ: Type) (A: type) (a: El A): El A := a.
Check type_idε. (* ∀ (ℰ: Type) (A: type), El A → El A. *)

Definition prop_idε (ℰ: Type) (A: Prop) (a: A): A := a.
Check prop_idε. (* ∀ (ℰ: Type) (A: Prop), A → A. *)

```

Note how the translation leaves term in the `Prop` hierarchy almost *intact*.

The real difference happens when translating inductives. The translation of propositional inductives result in a similar structure at the target theory, interpreting only computational terms as exceptional. A clear example is the existence inductive. The source inductive is the following

```

Inductive ex (A : Type) (P : A → Prop) : Prop :=
| ex_intro : ∀ (x: A), P x → ex A P.

```

The translated inductive is:

```

Inductive exε (ℰ: Type) (A : type) (P : El A → Prop) : Prop :=
| ex_introε : ∀ (x: El A), P x → exε ℰ A P.

```

Note that, the propositional part (**Prop** and **P x**) are left intact, while the computational ones are interpreted as exceptional (**Type** and **A**).

On the other hand, the translation of computational inductives are handled differently. Besides the translated inductive, they require a parametric inductive, and at the source and target theory, a catch induction, a parametric induction, an instance declaration and a consistency proof. The plugin generates all: the inductives, the instance of the type class and the proof of consistency. As an example, assume again the list inductive declaration.

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

Its exceptional translation corresponds to

```
Inductive listε (ℰ: Type) (A: type): Type :=
| nilε: listε ℰ A
| consε: El A → listε ℰ A → listε ℰ A.
| listφ: ℰ → listε ℰ A.
```

Note the added exceptional constructor. Its parametric exceptional translation corresponds to

```
Inductive list_param (ℰ: Type) (A: type): listε ℰ A → Type :=
| nil_param: list_param ℰ A (nilε ℰ A)
| cons_param: ∀ (a: El A) (l: listε ℰ A),
               list_param ℰ A l → list_param ℰ A (consε ℰ A a l).
```

The catch induction principle is declared as an axiom in the source theory:

```
Axiom list_catch:
  ∀ (A: Type) (P: list A → Type),
    P nil →
    (∀ (a: A) (l: list A), P l → P (cons a l)) →
    (∀ (e: ℰ), P (raise (list A) e)) →
    ∀ (l: list A), P l.
```

and it is mapped to the induction principle of the exceptional inductive. Similarly, the parametric induction principle is provided as an axiom in the source theory:

```
Axiom list_ind_param:
  ∀ (A: Type) (P: list A → Prop),
    P nil →
    (∀ (a: A) (l: list A), P l → P (cons a l)) →
    ∀ (l: list A), param l → P l.
```

but the target definition internally uses the induction principle of the parametric exceptional inductive.

The instance declaration of the modality in the source theory is also represented as an axiom.

```
Axiom Instance list_instance (A: Type): Param (list A).
```

The target instance is declared as follows

```
Instance list_instance $\varepsilon$  ( $\mathbb{E}$ : Type) (A: type):
  Param $\varepsilon$   $\mathbb{E}$  (TypeVal (list $\varepsilon$   $\mathbb{E}$  A) (list $\phi$   $\mathbb{E}$  A)) := {
    param $\varepsilon$  := list_param  $\mathbb{E}$  A;
    param_correct $\varepsilon$  := ...;
  }
```

The field `param_correct ε` can always be generated at translation time by an [inversion](#)². The result is a huge term that proves the statement.

5.6 Plugin Performance

The plugin’s performance for large projects is an open question and a formal assessment of it is out of scope of the current thesis. Nevertheless, we now explore some examples as an informal assessment.

First of all, we have to point out that there are two modes relevant to us when executing Coq: the interactive mode and the batch compilation mode. The user uses the interactive mode to write programs and theorems step by step. The batch compilation mode takes a file, usually written in interactive mode, and goes through it completely, executing commands and compiling terms [26]. For example, we can write the following definitions in a file in interactive mode:

```
Time Definition number := Eval compute in 80 * 505. (* 40400 *)
(* Finished transaction in 0.832 secs *)

Fixpoint slow_fixpoint (n m : nat) : nat.
Proof.
  do 20 (refine (id _)).
  destruct m.
  - exact 0.
  - exact (slow_fixpoint n m).
Time Defined.
(* Finished transaction in 2.49 secs *)

Time Definition fun_test (l: True) (r: nat) (l: True) (a: nat) :=
  id (id (id (id (id (id (id (id (id a)))))))).
(* Finished transaction in 0. secs *)
```

and we then can compile the file in batch execution mode. These terms are in pure Coq. The first command defines that `number` is equal to the value of `80 * 505`, which is 40400, and Coq takes 0.832 seconds to calculate such value. The second command defines a recursive function named `slow_fixpoint` through Coq’s tactic language [26] and Coq takes 2.49 seconds to validate the term (Coq does additional validations for recursive functions). The third command defines a term that Coq only has to type check. These examples illustrate use cases that happen when developing Coq programs, some of which can be time consuming.

²Coq tactic to derive equations on proofs

Let us come back to the plugin. The translation of RETT defined in Figure 3.3 establishes a conditional branch when translating a product or a term viewed as a type:

$$\begin{aligned}
[[\Pi(x : A). M]]_{\psi}^{\Gamma} &:= \begin{cases} \Pi(x : [[A]]_{\psi}^{\Gamma}). [[B]]_{\psi}^{\Gamma, x:A} & \Gamma \vdash \Pi(x : A). B : * \\ \text{TypeVal}_i & \sim \\ \Pi(x : [[A]]_{\psi}^{\Gamma}). [[B]]_{\psi}^{\Gamma, x:A} & \\ \lambda(e : \mathbb{E}) (x : [[A]]_{\psi}^{\Gamma}). [[B]]_{\psi\phi}^{\Gamma, x:A} e & \end{cases} \\
[[A]]_{\psi}^{\Gamma} &:= \begin{cases} [[A]]_{\psi}^{\Gamma} & \Gamma \vdash A : * \\ \text{El}_i [[A]]_{\psi}^{\Gamma} & \sim \end{cases}
\end{aligned}$$

The plugin has the same logic. This means that the plugin does type checking whenever it encounters one of the previous situations when translating a term. Here, it is important to highlight that the plugin performs type checking with verified Coq terms: terms where all computations and validations have already been performed. Translating the above terms yields:

```

Time Effect Translate number. (* Finished transaction in 0.451 secs *)
Time Effect Translate slow_fixpoint (* Finished transaction in 0.004 secs*)
Time Effect List Translate fun_test. (* Finished transaction in 0.004 secs*)

```

These numbers do not allow us to establish a relation between Coq definition time and plugin execution time. Additionally, the plugin generates terms when translating inductive types, but these terms are a syntactic derivation from the original inductive type, which should not produce an explosion in the plugin's execution time.

Nonetheless, these execution times can add up when executing Coq in batch mode. For example, the program presented in Section 5.1 can be considered small and its total compilation time takes about 0.43 seconds to compile in batch mode. If we take out all the non essential translations, the compilation takes 0.25 seconds in batch mode.

In this chapter we explored the implementation of CoqRETT, the implementation of RETT in Coq. We showed how it can be used, explained the internals of the plugin and made a brief analysis on its performance.

Chapter 6

Conclusion

In this thesis, we developed a dependent type theory with exceptions, which supports consistent reasoning about exceptional terms. Additionally, it is possible to enforce purity of exceptional terms when required.

The novel features of the theory are the logical consistency, which opens the door to develop certified software using exceptions, the fact that exceptions can be captured and transformed by the related structures of the theory, and that exceptional terms can be restricted through a modality.

At first glance, this work can be seen as a merge between two previous exceptional type theories: ETT and PETT. RETT provides exceptions as ETT and it is able to enforce purity on terms like PETT, although locally. In practice, RETT goes beyond. It allows reasoning about exceptions while remaining consistent, unlike ETT, and it is possible to enunciate theorems about exceptions, unlike PETT, while still being able to enforce purity.

Additionally, we developed a Coq plugin and were capable of proving theorems relating exceptional terms and propositions operating from the source theory, and checking their validity by applying the translation. This process comes with an overhead in execution and a formal assessment is required, but it seems negligible compared to Coq execution time at first glance.

The present type theory corresponds to an instance of the theory presented in the extended version in [19]. Here, the theory is closer to CIC having only two sorts to differentiate between pure and computational terms, while the extension has an additional sort that mediates between the other two. This difference allows the internalization of the parametricity predicate in the translation in the extended version, while here we require the use of the internal mechanism of type classes to lookup the corresponding parametricity predicate. Nevertheless, both works use the same implementation at the plugin level because this is where they coincide. The thesis's author proved the current theory valid and extended the plugin of [18] with missing features plus the extension to realize the current theory.

Finally, this work opens the door to a new world of possibilities. We were already capable of using exceptions in proof assistants, but with RETT, we have more control over them and we can trust the theorems that we prove. It is still cumbersome to work in CoqRETT due to

its implementation through axioms and the inability to extend reduction rules. Nevertheless, to our knowledge, it is the first implementation of an exceptional type theory with consistent reasoning.

Bibliography

- [1] H. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2), 1991.
- [2] J.-P. Bernardy, P. Jansson, and R. Paterson. Proofs for free: Parametricity for dependent types. *Journal of Functional Programming*, 22(02):107–152, 2012.
- [3] S. Boulier, P. Pédrot, and N. Tabareau. The next 700 syntactical models of type theory. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 182–194, New York, NY, USA, 2017. ACM.
- [4] C. Böhm and A. Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.
- [5] A. Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.
- [6] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.
- [7] A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.
- [8] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2):95 – 120, 1988.
- [9] H. Geuvers. Induction is not derivable in second order dependent type theory. In *Typed Lambda Calculi and Applications*, 2001.
- [10] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l’arithmétique d’ordre supérieur*. 1972.
- [11] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989.
- [12] W. A. Howard. The formulae-as-types notion of construction. 1969.
- [13] S. C. Kleene and J. B. Rosser. The inconsistency of certain formal logics. *Annals of Mathematics*, 1935.
- [14] G. Lee and B. Werner. Proof-irrelevant model of CC with predicative induction and judgmental equality. *Logical Methods in Computer Science*, 7(4), 2011.
- [15] Z. Luo. Ecc, an extended calculus of constructions. In *Fourth Annual Symposium on Logic in Computer Science*, pages 386–395, 1989.

- [16] C. Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In B. W. Paleo and D. Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, Jan. 2015.
- [17] P. Pédrot and N. Tabareau. An Effectful Way to Eliminate Addiction to Dependence. In *Logic in Computer Science (LICS)*, 2017.
- [18] P. Pédrot and N. Tabareau. Failure is Not an Option An Exceptional Type Theory. In *ESOP 2018 - 27th European Symposium on Programming*, LNCS, pages 245–271, Thessaloniki, Greece, 2018. Springer.
- [19] P. Pédrot, N. Tabareau, H. J. Fehrmann, and E. Tanter. A reasonably exceptional type theory. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.
- [20] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.
- [21] J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque Sur La Programmation*, pages 408–423, 1974.
- [22] B. Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30(3):222–262, 1908.
- [23] M. Sozeau and N. Oury. First-class type classes. In *Theorem Proving in Higher Order Logics*, pages 278–293, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [24] W. W. Tait. Intensional interpretations of functionals of finite type i. *The Journal of Symbolic Logic*, 32(2), 1967.
- [25] The Agda Team. *Agda documentation*, 2020 (accessed May, 2020). <https://agda.readthedocs.io/en/v2.6.1/>.
- [26] The Coq Development Team. The coq proof assistant, 2020. <https://doi.org/10.5281/zenodo.3744225>.
- [27] A. Timany and M. Sozeau. Consistency of the Predicative Calculus of Cumulative Inductive Constructions (pCuIC). Research Report RR-9105, KU Leuven, Belgium ; Inria Paris, Oct. 2017.
- [28] A. Timany and M. Sozeau. Cumulative Inductive Types in Coq. In *FSCD 2018 - 3rd International Conference on Formal Structures for Computation and Deduction*, Oxford, United Kingdom, July 2018.
- [29] P. Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, Nov. 2015.
- [30] M. Wenzel. *The Isabelle/Isar Reference Manual*, 2020 (accessed May, 2020). <https://isabelle.in.tum.de/dist/Isabelle2020/doc/isar-ref.pdf>.
- [31] B. Werner. *Une Théorie des Constructions Inductives*. Theses, Université Paris-Diderot - Paris VII, May 1994.