Failure is Not an Option
An Exceptional Type Theory

Pierre-Marie Pédrot[1(✉)] and Nicolas Tabareau[2]

[1] MPI-SWS, Saarbrücken, Germany
ppedrot@mpi-sws.org
[2] Inria, Nantes, France
nicolas.tabareau@inria.fr

Abstract. We define the *exceptional translation*, a syntactic translation of the Calculus of Inductive Constructions (CIC) into itself, that covers full dependent elimination. The new resulting type theory features call-by-name exceptions with decidable type-checking and canonicity, but at the price of inconsistency. Then, noticing parametricity amounts to Kreisel's realizability in this setting, we provide an additional layer on top of the exceptional translation in order to tame exceptions and ensure that all exceptions used locally are caught, leading to the *parametric exceptional translation* which fully preserves consistency. This way, we can consistently extend the logical expressivity of CIC with independence of premises, Markov's rule, and the negation of function extensionality while retaining η-expansion. As a byproduct, we also show that Markov's principle is not provable in CIC. Both translations have been implemented in a Coq plugin, which we use to formalize the examples.

1 Introduction

Monadic translations constitute a canonical way to add effects to pure functional languages [1]. Until recently, this technique was not available for type theories such as CIC because of complex interactions with dependency. In a recent paper [2], we have presented a generic way to extend the monadic translation to dependent types, using the *weaning translation*, as soon as the monad under consideration satisfies a crucial property: being self-algebraic. Indeed, in the same way that the universe of types $\square_i$ is itself a type (of a higher universe) in type theory, the type of algebras of a monad T

$$\Sigma A : \square_i.\, \mathrm{T}\ A \to A$$

needs to be itself an algebra of the monad to allow a correct translation of the universe. However, in general, the weaning translation does not interpret all of CIC because dependent elimination needs to be restricted to linear predicates, that is, those that are intuitively call-by-value [3]. In this paper, we study the particular case of the error monad, and show that its weaning translation can be simplified and tweaked so that full dependent elimination is valid.

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 245–271, 2018.
https://doi.org/10.1007/978-3-319-89884-1_9

This *exceptional translation* gives rise to a novel extension of CIC with new computational behaviours, namely call-by-name exceptions.[1] That is, the type theory induced by the exceptional translation features new operations to raise and catch exceptions. This new logical expressivity comes at a cost, as the resulting theory is not consistent anymore, although still being computationally relevant. This means that it is possible to prove a contradiction, but, thanks to a weak form of canonicity, only because of an unhandled exception. Furthermore, the translation allows us to reason directly in CIC on terms of the exceptional theory, letting us prove, e.g., that assuming some properties on its input, an exceptional function actually never raises an exception. We thus have a sound logical framework to prove safety properties about impure dependently-typed programs.

We then push this technique further by noticing that parametricity provides a systematic way to describe that a term is not allowed to produce uncaught exceptions, bridging the gap between Kreisel's modified realizability [4] and parametricity inside type theory [5]. This *parametric exceptional translation* ensures that no exception reaches toplevel, thus ensuring consistency of the resulting theory. Pure terms are automatically handled, while it is necessary to show parametricity manually for terms internally using exceptions. We exploit this computational extension of CIC to show various logical results over CIC.

Contributions

- We describe the *exceptional translation*, the first monadic translation for the error monad for CIC, including strong elimination of inductive types, resulting in a sound logical framework to reason about impure dependently-typed programs.
- We use parametricity to extend the exceptional translation, getting a consistent variant dubbed the *parametric exceptional translation*.
- We show that Markov's rule is admissible in CIC.
- We show that definitional η-expansion together with the negation of function extensionality is admissible in CIC.
- We show that there exists a syntactical model of CIC that validates the independence of premises (which is known to be generally not valid in intuitionistic logic [6]) and use it to recover the recent result of Coquand and Mannaa [7], *i.e.*, that Markov's principle is not provable in CIC.
- We provide a Coq plugin[2] that implements both translations and with which we have formalized all the examples.

Plan of the Paper. In Sect. 2, we describe the exceptional translation and the resulting new computational principles arising from it. In Sect. 3, we present the parametric variant of the exceptional translation. Section 4 is devoted to the

[1] The fact that the resulting exception are call-by-name is explained in detailed in [2] using a call-by-push-value decomposition. Intuitively, it comes from the fact that CIC is naturally call-by-name.

[2] The plugin is available at https://github.com/CoqHott/exceptional-tt.

$$A, B, M, N ::= \Box_i \mid x \mid M\ N \mid \lambda x : A.\, M \mid \Pi x : A.\, B$$

$$\Gamma, \Delta ::= \cdot \mid \Gamma, x : A$$

$$\frac{\vdash \Gamma \qquad i < j}{\Gamma \vdash \Box_i : \Box_j} \qquad \frac{\Gamma \vdash M : B \qquad \Gamma \vdash A : \Box_i}{\Gamma, x : A \vdash M : B}$$

$$\frac{\Gamma \vdash A : \Box_i \qquad \Gamma, x : A \vdash B : \Box_j}{\Gamma \vdash \Pi x : A.\, B : \Box_{\max(i,j)}} \qquad \frac{\Gamma \vdash M : B \qquad \Gamma \vdash A : \Box_i \qquad A \equiv B}{\Gamma \vdash M : A}$$

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

$$\frac{}{\vdash \cdot} \qquad \frac{\Gamma \vdash A : \Box_i}{\vdash \Gamma, x : A} \qquad \frac{\Gamma \vdash A : \Box_i}{\Gamma, x : A \vdash x : A}$$

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

Fig. 1. Typing rules of CC_ω

various logical results resulting from the parametric exceptional translations. In Sect. 5, we discuss possible extensions of the translation with negative records and an impredicative universe. Section 6 describes the Coq plugin and illustrates its use on a concrete example. We discuss related work in Sect. 7 and conclude in Sect. 8.

2 The Exceptional Translation

We define in this section the exceptional translation as a syntactic translation between type theories. We call the target theory $\mathcal{T}$, upon which we will make various assumptions depending on the objects we want to translate.

2.1 Adding Exceptions to CC_ω

In this section, we describe the exceptional translation over a purely negative theory, *i.e.*, featuring only universes and dependent functions, called CC_ω, which is presented in Fig. 1. This theory is a predicative version of the Calculus of Constructions [8], with an infinite hierarchy of universes $\Box_i$ instead of one impredicative sort. We assume from now on that $\mathcal{T}$ contains at least CC_ω itself.

The exceptional translation is a simplification of the weaning translation [2] applied to the error monad. Owing to the fact that it is specifically tailored for exceptions, this allows to give a more compact presentation of it.

Let $\mathbb{E} : \Box_0$ be a fixed type of exceptions in $\mathcal{T}$. The weaning translation for the error monad amounts to interpret types as algebras, *i.e.*, as inhabitants of

the dependent sum $\Sigma A : \square_i. (A + \mathbb{E}) \to A$. In this paper, we take advantage of the fact that the algebra morphism restricted to A is always the identity. Thus every type just comes with a way to interpret failure on this type, i.e. types are intuitively interpreted as a pair of an $A : \square_i$ with a default (raise) function $A_\varnothing : \mathbb{E} \to A$. In practice, it is slightly more complicated as the universe of types itself is a type, so its interpretation must comes with a default function. We overcome this issue by assuming a term $\mathtt{type}_i$, representing types that can raise exceptions. This type comes with two constructors: $\mathtt{TypeVal}_i$ which allows to construct a $\mathtt{type}_i$ from a type and a default function on this type ; and another constructor $\mathtt{TypeErr}_i$ that represents the default function at the level of $\mathtt{type}_i$. Furthermore, $\mathtt{type}_i$ is equipped with an eliminator $\mathtt{type\_elim}_i$ and thus can be thought of as an inductive definition. For simplicity, we axiomatize it instead of requiring inductive types in the target of the translation.

Definition 1. *We assume that $\mathcal{T}$ features the data below, where i, j indices stand for universe polymorphism.*

- $\Omega_i : \mathbb{E} \to \square_i$
- $\omega_i : \Pi e : \mathbb{E}. \Omega_i\ e$
- $\mathtt{type}_i : \square_j$, *where* $i < j$
- $\mathtt{TypeVal}_i : \Pi A : \square_i. (\mathbb{E} \to A) \to \mathtt{type}_i$
- $\mathtt{TypeErr}_i : \mathbb{E} \to \mathtt{type}_i$
- $\mathtt{type\_elim}_{i,j} :\ \Pi P : \mathtt{type}_i \to \square_j.$
  $(\Pi (A : \square_i)\, (A_\varnothing : \mathbb{E} \to A). P\ (\mathtt{TypeVal}_i\ A\ A_\varnothing)) \to$
  $(\Pi e : \mathbb{E}. P\ (\mathtt{TypeErr}_i\ e)) \to \Pi T : \mathtt{type}_i. P\ T$

subject to the following definitional equations:

$$\mathtt{type\_elim}_{i,j}\ P\ p_v\ p_\varnothing\ (\mathtt{TypeVal}_i\ A\ A_\varnothing) \equiv p_v\ A\ A_\varnothing$$

$$\mathtt{type\_elim}_{i,j}\ P\ p_v\ p_\varnothing\ (\mathtt{TypeErr}_i\ e) \equiv p_\varnothing\ e$$

The Ω term describes what it means for a type to fail, i.e. it ascribes a meaning to sequents of the form $\Gamma \vdash M : \mathtt{fail}\ e$. In practice, it is irrelevant and can be chosen to be degenerate, e.g. $\Omega := \lambda\_ : \mathbb{E}. \mathtt{unit}$.

In what follows, we often leave the universe indices implicit although they can be retrieved at the cost of more explicit annotations.

Before defining the exceptional translation we need to derive a term $\mathtt{El}$[3] that recovers the underlying type from an inhabitant of $\mathtt{type}$ and $\mathtt{Err}$ that lifts the default function to this underlying type.

Definition 2. *From the data of Definition 1, we derive the following terms.*

$$\begin{aligned}
\mathtt{El}_i\ &:\ \ \mathtt{type}_i \to \square_i \\
&:= \lambda A : \mathtt{type}_i. \mathtt{type_elim}\ (\lambda T : \mathtt{type}_i. \square_i) \\
&\qquad (\lambda (A_0 : \square_i)\, (A_\varnothing : \mathbb{E} \to A_0). A_0)\ \Omega\ A \\
\mathtt{Err}_i\ &:\ \ \Pi A : \mathtt{type}_i. \mathbb{E} \to \mathtt{El}_i\ A \\
&:= \lambda (A : \mathtt{type}_i)\, (e : \mathbb{E}). \mathtt{type_elim}\ \mathtt{El}_i \\
&\qquad (\lambda (A_0 : \square_i)\, (A_\varnothing : \mathbb{E} \to A_0). A_\varnothing\ e)\ \omega\ A
\end{aligned}$$

[3] The notation $\mathtt{El}$ refers to universes à la Tarski in Martin-Löf type theory.

$$
\begin{aligned}
[\Box_i] &:= \texttt{TypeVal}\ \texttt{type}_i\ \texttt{TypeErr}_i \\
[x] &:= x \\
[\lambda x : A.\, M] &:= \lambda x : [\![A]\!].\, [M] \\
[M\ N] &:= [M]\ [N] \\
[\Pi x : A.\, B] &:= \texttt{TypeVal}\ (\Pi x : [\![A]\!].\, [\![B]\!])\ (\lambda (e : \mathbb{E})\, (x : [\![A]\!]).\, [B]_{\varnothing}\ e) \\
[A]_{\varnothing} &:= \texttt{Err}\ [A] \\
[\![A]\!] &:= \texttt{El}\ [A] \\
[\![\cdot]\!] &:= \cdot \\
[\![\Gamma, x : A]\!] &:= [\![\Gamma]\!], x : [\![A]\!]
\end{aligned}
$$

Fig. 2. Exceptional translation

The exceptional translation is defined in Fig. 2. As usual for syntactic translations [9], the term translation is given by $[\cdot]$ and the type translation, written $[\![\cdot]\!]$, is derived from it using the function `El`. There is an additional macro $[\cdot]_{\varnothing}$, defined using $\texttt{Err}_i$, which corresponds to the way to inhabit a given type from an exception.

Note that we will often slightly abuse the translation and use the $[\cdot]$ and $[\![\cdot]\!]$ notation as macros acting on the target theory. This is merely for readability purposes, and the corresponding uses are easily expanded to the actual term.

The following lemma makes explicit how $[\![\cdot]\!]$ and $[\cdot]_{\varnothing}$ behave on universes and on the dependent function space.

Lemma 3 (Unfoldings). *The following definitional equations hold:*

- $[\![\Box_i]\!] \equiv \texttt{type}_i$
- $[\![\Pi x : A.\, B]\!] \equiv \Pi x : [\![A]\!].\, [\![B]\!]$
- $[\Box_i]_{\varnothing}\ e \equiv \texttt{TypeErr}_i\ e$
- $[\Pi x : A.\, B]_{\varnothing}\ e \equiv \lambda x : [\![A]\!].\, [B]_{\varnothing}\ e$

Proof. By unfolding and straightforward reductions.

The soundness of the translation follows from the following properties, which are fundamental but straightforward to prove.

Theorem 4 (Soundness). *The following properties hold.*

- $[M\{x := N\}] \equiv [M]\{x := [N]\}$ *(substitution lemma).*
- *If* $M \equiv N$ *then* $[M] \equiv [N]$ *(conversion lemma).*
- *If* $\Gamma \vdash M : A$ *then* $[\![\Gamma]\!] \vdash [M] : [\![A]\!]$ *(typing soundness).*
- *If* $\Gamma \vdash A : \Box$ *then* $[\![\Gamma]\!] \vdash [A]_{\varnothing} : \mathbb{E} \to [\![A]\!]$ *(exception soundness).*

Proof. The first property is by routine induction on M, the second is direct by induction on the conversion derivation. The third is by induction on the

typing derivation, the most important rule being $\square_i : \square_j$, which holds because $[\square_i] \equiv \texttt{TypeVal type}_i\ \texttt{TypeErr}_i$ has type $\texttt{type}_j$ which is convertible to $[\![\square_j]\!]$ by Lemma 3. The last property is a direct application of typing soundness and unfolding of Lemma 3 for universes.

We call $\mathcal{T}_{\mathbb{E}}$ the theory arising from this interpretation, which is formally defined in a way similar to standard categorical constructions over dependent type theory. Terms and contexts of $\mathcal{T}_{\mathbb{E}}$ are simply terms and contexts of $\mathcal{T}$. A context Γ is valid is $\mathcal{T}_{\mathbb{E}}$ whenever its translation $[\![\Gamma]\!]$ is valid in $\mathcal{T}$. Two terms M and N are convertible in $\mathcal{T}_{\mathbb{E}}$ whenever their translations $[M]$ and $[N]$ are convertible in $\mathcal{T}$. Finally, $\Gamma \vdash_{\mathcal{T}_{\mathbb{E}}} M : A$ whenever $[\![\Gamma]\!] \vdash_{\mathcal{T}} [M] : [\![A]\!]$.

That is, it is possible to extend $\mathcal{T}_{\mathbb{E}}$ with a new constant $\texttt{c}$ of a given type A by providing an inhabitant $\texttt{c}_{\mathbb{E}}$ of the translated type $[\![A]\!]$. Then the translation is extended with $[\texttt{c}] := \texttt{c}_{\mathbb{E}}$. The potential computational rules satisfied by this new constant are directly given by the computational rules satisfied by its translation. In some sense, the new constant $\texttt{c}$ is just syntactic sugar for $\texttt{c}_{\mathbb{E}}$. Using $\mathcal{T}_{\mathbb{E}}$, Theorem 4 can be rephrased in the following way.

Theorem 5. *If $\mathcal{T}$ interprets CC_ω then so does $\mathcal{T}_{\mathbb{E}}$, that is, the exceptional translation is a syntactic model of CC_ω.*

2.2 Exceptional Inductive Types

The fact that the only effect we consider is raising exceptions does not really affect the negative fragment when compared to our previous work [2], but it sure shines when it comes to interpreting inductive datatypes. Indeed, as explained in the introduction, the weaning translation only interprets a subset of CIC, restricting dependent elimination to linear predicates. Furthermore, it also requires a few syntactic properties of the underlying monad ensuring that positivity criteria are preserved through the translation, which can be sometimes hard to obtain.

The exceptional translation diverges from the weaning translation precisely on inductives types. It allows a more compact translation of the latter, while at the same time providing a complete interpretation of CIC, that is, including full dependent elimination.

From now on, we assume that the target theory is a predicative restriction of CIC, i.e. that we can construct in it new inductive datatypes as we do in e.g. Coq [10], but without considering an impredicative universe. That is, all the inductive types we consider in this section live in $\square$. As a matter of fact, we slightly abuse the usual nomenclature and simply call CIC this predicative fragment in the remainder of the paper. We refrain from describing the generic typing rules that extend CC_ω into CIC, as they are fairly standard and would take up too much space. See for instance Werner's thesis for a comprehensive presentation [11].

$$
\begin{aligned}
&[\mathcal{I}] \;:= \lambda(p_1 : [\![P_1]\!]) \;\ldots\; (p_n : [\![P_n]\!])\,(i_1 : [\![I_1]\!]) \;\ldots\; (i_m : [\![I_m]\!]).\\
&\qquad\quad \texttt{TypeVal}\;(\mathcal{I}^\bullet\; p_1 \;\ldots\; p_n\; i_1 \;\ldots\; i_m)\;(\mathcal{I}_\varnothing\; p_1 \;\ldots\; p_n\; i_1 \;\ldots\; i_m)\\
&[c_1] := c_1^\bullet\\
&\ldots\\
&[c_k] := c_k^\bullet
\end{aligned}
$$

Fig. 3. Inductive type translation

Type and Constructor Translation. As explained before, the intuitive interpretation of a type through the exceptional translation is a pair of a type and a default function from exceptions into that type. In particular, when translating some inductive type $\mathcal{I}$, we must come up with a type $[\![\mathcal{I}]\!]$ together with a default function $\mathbb{E} \to [\![\mathcal{I}]\!]$. As soon as $\mathbb{E}$ is inhabited, that means that we need $[\![\mathcal{I}]\!]$ to be inhabited, preferably in a canonical way. The solution is simple: just as for types where we freely added the exceptional case by means of the `TypeErr` constructor, we freely add exceptions to every inductive type.

In practice, there is an elegant and simple way to do this. It just consists in translating constructors pointwise, while adding a new dedicated constructor standing for the exceptional case. We now turn to the formal construction.

Definition 6. *Let $\mathcal{I}$ be an inductive datatype with*

- *parameters* $p_1 : P_1, \ldots, p_n : P_n$;
- *indices* $i_1 : I_1, \ldots, i_m : I_m$;
- *constructors*

$$
\begin{aligned}
&c_1 : \Pi(a_{1,1} : A_{1,1}) \ldots (a_{1,l_1} : A_{1,l_1}).\, \mathcal{I}\; p_1 \;\ldots\; p_n\; V_{1,1} \;\ldots\; V_{1,m}\\
&\ldots\\
&c_k : \Pi(a_{k,1} : A_{k,1}) \ldots (a_{k,l_k} : A_{k,l_k}).\, \mathcal{I}\; p_1 \;\ldots\; p_n\; V_{k,1} \;\ldots\; V_{k,m}
\end{aligned}
$$

We define the exceptional translation of $\mathcal{I}$ and its constructors in Fig. 3, where $\mathcal{I}^\bullet$ is the inductive type defined by

- *parameters* $p_1 : [\![P_1]\!], \ldots, p_n : [\![P_n]\!]$;
- *indices* $i_1 : [\![I_1]\!], \ldots, i_m : [\![I_m]\!]$;
- *constructors*

$$
\begin{aligned}
&c_1^\bullet : \Pi(a_{1,1} : [\![A_{1,1}]\!]) \ldots (a_{1,l_1} : [\![A_{1,l_1}]\!]).\, \mathcal{I}^\bullet\; p_1 \;\ldots\; p_n\; [V_{1,1}] \;\ldots\; [V_{1,m}]\\
&\ldots\\
&c_k^\bullet : \Pi(a_{k,1} : [\![A_{k,1}]\!]) \ldots (a_{k,l_k} : [\![A_{k,l_k}]\!]).\, \mathcal{I}^\bullet\; p_1 \;\ldots\; p_n\; [V_{k,1}] \;\ldots\; [V_{k,m}]\\
&\mathcal{I}_\varnothing : \Pi(i_1 : [\![I_1]\!]) \ldots (i_m : [\![I_m]\!]).\, \mathbb{E} \to \mathcal{I}^\bullet\; p_1 \;\ldots\; p_n\; i_1 \;\ldots\; i_m
\end{aligned}
$$

where in the recursive calls in the various A, we locally set

$$[\![\mathcal{I}\; M_1 \;\ldots\; M_n\; N_1 \;\ldots\; N_m]\!] := \mathcal{I}^\bullet\; [M_1] \;\ldots\; [M_n]\; [N_1] \;\ldots\; [N_m].$$

Example 7. We give a few representative examples of the inductive translation in Fig. 4 in a Coq-like syntax. They were chosen because they are simple instances of inductive types featuring parameters, indices and recursion in an orthogonal way. For convenience, we write $\Sigma\; A\; (\lambda x : A.\, B)$ as $\Sigma x : A.\, B$.

$$
\begin{array}{l}
\texttt{Ind bool} : \square := \\
\quad \mid \texttt{true} : \texttt{bool} \\
\quad \mid \texttt{false} : \texttt{bool} \\
\\
\texttt{Ind list}\ (A : \square) : \square := \\
\quad \mid \texttt{nil} : \texttt{list}\ A \\
\quad \mid \texttt{cons} : A \to \texttt{list}\ A \to \texttt{list}\ A \\
\\
\texttt{Ind}\ \Sigma\ (A : \square)\,(B : A \to \square) : \square := \\
\quad \mid \texttt{ex} : \Pi(x : A)\,(y : B\ x).\ \Sigma\ A\ B \\
\\
\texttt{Ind eq}\ (A : \square)\,(x : A) : A \to \square := \\
\quad \mid \texttt{refl} : \texttt{eq}\ A\ x\ x
\end{array}
\qquad
\begin{array}{l}
\texttt{Ind bool}^\bullet : \square := \\
\quad \mid \texttt{true}^\bullet : \texttt{bool}^\bullet \\
\quad \mid \texttt{false}^\bullet : \texttt{bool}^\bullet \\
\quad \mid \texttt{bool}_\varnothing : \mathbb{E} \to \texttt{bool}^\bullet \\
\texttt{Ind list}^\bullet\ (A : [\![\square]\!]) : \square := \\
\quad \mid \texttt{nil}^\bullet : \texttt{list}^\bullet\ A \\
\quad \mid \texttt{cons}^\bullet : [\![A]\!] \to \texttt{list}^\bullet\ A \to \texttt{list}^\bullet\ A \\
\quad \mid \texttt{list}_\varnothing : \mathbb{E} \to \texttt{list}^\bullet\ A \\
\texttt{Ind}\ \Sigma^\bullet\ (A : [\![\square]\!])\,(B : [\![A]\!] \to \square) : \square := \\
\quad \mid \texttt{ex}^\bullet : \Pi(x : [\![A]\!])\,(y : [\![B\ x]\!]).\ \Sigma^\bullet\ A\ B \\
\quad \mid \Sigma_\varnothing : \mathbb{E} \to \Sigma^\bullet\ A\ B \\
\texttt{Ind eq}^\bullet\ (A : [\![\square]\!])\,(x : [\![A]\!]) : [\![A]\!] \to \square := \\
\quad \mid \texttt{refl}^\bullet : \texttt{eq}^\bullet\ A\ x\ x \\
\quad \mid \texttt{eq}_\varnothing : \Pi y : [\![A]\!].\ \mathbb{E} \to \texttt{eq}^\bullet\ A\ x\ y
\end{array}
$$

Fig. 4. Examples of translations of inductive types

Remark 8. The fact the we locally override the translation for recursive calls on the $[\![\cdot]\!]$ translation of the type being defined means that we cannot handle cases where the translation of the type of a constructor actually contains an instance of $[\mathcal{I}]$. Because of the syntactic positivity criterion, the only possibility for such a situation to occur in CIC is in the so-called nested inductive definitions. However, nested inductive types are essentially a programming convenience, as most nested types can be rewritten in an isomorphic way that is not nested.

Lemma 9. *If $\mathcal{I}$ is given as in Definition 6, we have for any terms $\vec{M}$, $\vec{N}$*

$$[\![\mathcal{I}\ M_1\ \ldots\ M_n\ N_1\ \ldots\ N_m]\!] \equiv \mathcal{I}^\bullet\ [M_1]\ \ldots\ [M_n]\ [N_1]\ \ldots\ [N_m].$$

This justifies a posteriori the simplified local definition we used in the recursive calls of the translation of the constructors.

Theorem 10. *For any inductive type $\mathcal{I}$ not using nested inductive types, the translation from Definition 6 is well-typed and satisfies the positivity criterion.*

Proof. Preservation of typing is a consequence of Theorem 4. The restriction on nested types, which is slightly stronger than the usual positivity criterion of CIC, is due to the fact that $\mathcal{I}_\varnothing$ is not available in the recursive calls and thus cannot be used to build a term of type `type` via the `TypeVal` constructor.

Preservation of the positivity criterion is straightforward, as the shape of every constructor c_k is preserved, and furthermore by Lemma 3 the structure of every argument type is preserved by $[\![\cdot]\!]$ as well. The only additional constructor $\mathcal{I}_\varnothing$ does not mention the recursive type and is thus automatically positive.

Corollary 11. *Type soundness holds for the translation of inductive types and their constructors.*

Pattern-Matching Translation. We now turn to the translation of the elimination of inductive terms, that is, pattern matching. Once again, its definition originates from the fact that we are working with call-by-name exceptions. It is well-known that in call-by-name, pattern matching implements a delimited form of call-by-value, by forcing its scrutinee before proceeding, at least up to the head constructor. Therefore, as soon as the matched term (re-)raises an exception, the whole pattern-matching reraises the same exception. A little care has to be taken in order to accomodate for the fact that the return type of the pattern-matching depends on the scrutinee, in particular when it is the default constructor of the inductive type.

In what follows, we use the $i_1 \ldots i_n$ notation for clarity, but compact it to $\vec{i}$ for space reasons, when appropriate.

Definition 12. *Assume an inductive $\mathcal{I}$ as given in Definition 6. Let Q be the well-typed pattern-matching defined as*

$$\begin{array}{l}\texttt{match}\ M\ \texttt{return}\ \lambda(i_1 : I_1) \ldots (i_m : I_m)\,(x : \mathcal{I}\ X_1\ \ldots\ X_n\ i_1\ \ldots\ i_m).\, R\ \texttt{with}\\ \mid\ c_1\ a_{1,1}\ \ldots\ a_{1,l_1}\ \Rightarrow\ N_1\\ \ldots\\ \mid\ c_k\ a_{k,1}\ \ldots\ a_{k,l_k}\ \Rightarrow\ N_k\\ \texttt{end}\end{array}$$

where

$$\Gamma \vdash \vec{X} : \vec{P} \qquad \Gamma \vdash \vec{Y} : \vec{I}\{\vec{p} := \vec{X}\} \qquad \Gamma \vdash M : \mathcal{I}\ X_1\ \ldots\ X_n\ Y_1\ \ldots\ Y_m$$
$$\Gamma, \vec{i} : \vec{I}\{\vec{p} := \vec{X}\}, x : \mathcal{I}\ \vec{X}\ \vec{i} \vdash R : \square \qquad \Gamma \vdash Q : R\{\vec{i} := \vec{Y}, x := M\}$$
$$\Gamma, \vec{a}_1 : \vec{A}_1 \vdash N_1 : R\{\vec{i} := \vec{V}_1\{\vec{p} := \vec{X}\}, x := c_1\ \vec{X}\ \vec{a}_1\}$$
$$\ldots$$
$$\Gamma, \vec{a}_k : \vec{A}_k \vdash N_k : R\{\vec{i} := \vec{V}_k\{\vec{p} := \vec{X}\}, x := c_k\ \vec{X}\ \vec{a}_k\}$$

then we pose $[Q]$ to be the following pattern-matching.

$$\begin{array}{l}\texttt{match}\ [M]\ \texttt{return}\ \lambda(i_1 : [\![I_1]\!]) \ldots (i_m : [\![I_m]\!])\,(x : \mathcal{I}^\bullet\ [X_1] \ldots [X_n]\ i_1 \ldots i_m).\, [\![R]\!]\ \texttt{with}\\ \mid\ c_1^\bullet\ a_{1,1}\ \ldots\ a_{1,l_1}\ \Rightarrow\ [N_1]\\ \ldots\\ \mid\ c_k^\bullet\ a_{k,1}\ \ldots\ a_{k,l_k}\ \Rightarrow\ [N_k]\\ \mid\ \mathcal{I}_\varnothing\ i_1\ \ldots\ i_m\ e\ \Rightarrow\ [R]_\varnothing\{x := \mathcal{I}_\varnothing\ X_1\ \ldots\ X_n\ i_1\ \ldots\ i_m\ e\}\ e\\ \texttt{end}\end{array}$$

Lemma 13. *With notations and typing assumptions from Definition 12, we have*

$$[\![\Gamma]\!] \vdash [Q] : [\![R]\!]\{\vec{i} := [\vec{Y}], x := [M]\}.$$

Proof. Mostly a consequence of Theorem 4 applied to all of the premises of the pattern-matching rule. The only thing we have to check specifically is that the branch for the default constructor $\mathcal{I}_\varnothing$ is well-typed as

$$[\![\Gamma]\!], \vec{i} : \vec{I}\{\vec{p} := \vec{X}\}, e : \mathbb{E} \vdash [R]_\varnothing\{x := \mathcal{I}_\varnothing\ \vec{X}\ \vec{i}\ e\}\ e : [\![R]\!]\{x := \mathcal{I}_\varnothing\ \vec{X}\ \vec{i}\ e\}$$

which is also due to Theorem 4 applied to R.

Lemma 14. *The translation preserves ι-rules.*

Proof. Immediate, as the translation preserves the structure of the patterns.

The translation is also applicable to fixpoints, but for the sake of readability we do not want to fully spell it out, although it is simply defined by congruence (commutation with the syntax). As such, it trivially preserves typing and reduction rules. Note that the Coq plugin presented in Sect. 6 features a complete translation of inductive types, pattern-matching and fixpoints. So the interested reader may experiment with the plugin to see how fixpoints are translated.

Therefore, by summarizing all of the previous properties, we have the following result.

Theorem 15. *If $\mathcal{T}$ interprets* CIC*, then so does $\mathcal{T}_{\mathbb{E}}$, and thus the exceptional translation is a syntactic model of* CIC*.*

2.3 Flirting with Inconsistency

It is now time to point at the elephant in the room. The exceptional translation has a lot of nice properties, but it has one grave defect.

Theorem 16. *If $\mathbb{E}$ is inhabited, then $\mathcal{T}_{\mathbb{E}}$ is logically inconsistent.*

Proof. The empty type is translated as

$$\texttt{Ind empty}^{\bullet} : \square := \texttt{empty}_{\varnothing} : \mathbb{E} \to \texttt{empty}^{\bullet}$$

which is inhabited as soon as $\mathbb{E}$ is.

Note that when $\mathbb{E}$ is empty, the situation is hardly better, as the translation is essentially the identity. However, when $\mathcal{T}$ satisfies canonicity, the situation is not totally desperate as $\mathcal{T}_{\mathbb{E}}$ enjoys the following weaker canonicity lemma.

Lemma 17 (Exceptional Canonicity). *Let $\mathcal{I}$ be an inductive type with constructors $c_1, \ldots, c_n$ and assume that $\mathcal{T}$ satisfies canonicity. The translation of any closed term $\vdash_{\mathcal{T}_{\mathbb{E}}} M : \mathcal{I}$ evaluates either to a constructor of the form $c_i^{\bullet}\ N_1 \ldots N_{l_i}$ or to the default constructor $\mathcal{I}_{\varnothing}\ e$ for some $e : \mathbb{E}$.*

Proof. Direct application of Theorem 4 and canonicity of $\mathcal{T}$.

A direct consequence of Lemma 17 is that any proof of the empty type is an exception. As we will see in Sect. 4.1, for some types it is also possible to dynamically check whether a term of this type is a correct proof, in the sense that it does not raise an uncaught exception. This means that while $\mathcal{T}_{\mathbb{E}}$ is logically unsound, it is computationally relevant and can still be used as a *dependently-typed programming language with exceptions*, a shift into a realm where we would have called the weaker canonicity Lemma 17 a *progress lemma*.

This is not the end of the story, though. Recall that $\mathcal{T}_{\mathbb{E}}$ only exists through its embedding $[\cdot]$ into $\mathcal{T}$. In particular, if $\mathcal{T}$ is consistent, this means that one can reason about terms of $\mathcal{T}_{\mathbb{E}}$ directly in $\mathcal{T}$. For instance, it is possible to prove

in $\mathcal{T}$ that assuming some properties about its input, a function in $\mathcal{T}_{\mathbb{E}}$ never raises an exception. Hence not only do we have an effectul programming language, but we also have a *sound logical framework* allowing to transparently prove safety properties about impure programs.

It is actually even better than that. We will show in Sect. 3 that safety properties can be derived automatically for pure programs, allowing to recover a consistent type theory as long as $\mathcal{T}$ is consistent itself.

2.4 Living in an Exceptional World

We describe here what $\mathcal{T}_{\mathbb{E}}$ feels like in direct style. The exceptional theory feature a new type $\mathbf{E}$ which reifies the underlying type $\mathbb{E}$ of exceptions in $\mathcal{T}_{\mathbb{E}}$. It uses the fact that for $\mathbb{E}$, the default function (here of type $\mathbb{E} \to \mathbb{E}$) can simply be defined as the identity function. Its translation is given by

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

Then, it is possible to define in $\mathcal{T}_{\mathbb{E}}$ a function $\texttt{raise} : \Pi A : \Box.\, \mathbf{E} \to A$ that raises the provided exception at any type as

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

As we have already mentioned, the reader should be aware that the exceptions arising from this translation are call-by-name. This means that they do not behave like their usual call-by-value counterpart. In particular, we have in $\mathcal{T}_{\mathbb{E}}$

$$\texttt{raise}\ (\Pi x : A.\, B)\ e \equiv \lambda x : A.\, \texttt{raise}\ B\ e$$

which means that exceptions cannot be caught on Π-types. We can catch them on universes and inductive types though, because in those cases they are freely added through an extra constructor which one can pattern-match on. For instance, there exists in $\mathcal{T}_{\mathbb{E}}$ a term

$$\begin{aligned}\texttt{catch}_{\texttt{bool}} : \Pi P : \texttt{bool} \to \Box.\, P\ \texttt{true} \to P\ \texttt{false} \to \\ (\Pi e : \mathbf{E}.\, P\ (\texttt{raise bool}\ e)) \to \Pi b : \texttt{bool}.\, P\ b\end{aligned}$$

defined by

$$\begin{aligned}[\texttt{catch}_{\texttt{bool}}] := \lambda P\, p_t\, p_f\, p_e\, b.\ \texttt{match}\ b\ \texttt{return}\ \lambda b.\, \texttt{El}\ (P\ b)\ \texttt{with} \\ \mid \texttt{true}^{\bullet} \Rightarrow p_t \\ \mid \texttt{false}^{\bullet} \Rightarrow p_f \\ \mid \texttt{bool}_{\varnothing}\ e \Rightarrow p_e\ e \\ \texttt{end}\end{aligned}$$

satisfying the expected reduction rules on all three cases.

In Sect. 6, we illustrate the use of the exceptional theory using the Coq plugin to define a simple cast framework as in [12].

$$
\begin{aligned}
[\Box_i]_\varepsilon &:= \lambda A : [\![\Box_i]\!].\, [\![A]\!] \to \Box_i \\
[x]_\varepsilon &:= x_\varepsilon \\
[\lambda x : A.\, M]_\varepsilon &:= \lambda (x : [\![A]\!])\, (x_\varepsilon : [\![A]\!]_\varepsilon\ x).\, [M]_\varepsilon \\
[M\ N]_\varepsilon &:= [M]_\varepsilon\ [N]\ [N]_\varepsilon \\
[\Pi x : A.\, B]_\varepsilon &:= \lambda (f : \Pi x : [\![A]\!].\, [\![B]\!]).\, \Pi (x : [\![A]\!])\, (x_\varepsilon : [\![A]\!]_\varepsilon\ x).\, [\![B]\!]_\varepsilon\ (f\ x) \\
[\![A]\!]_\varepsilon &:= [A]_\varepsilon \\
[\![\cdot]\!]_\varepsilon &:= \cdot \\
[\![\Gamma, x : A]\!]_\varepsilon &:= [\![\Gamma]\!]_\varepsilon, x : [\![A]\!], x_\varepsilon : [\![A]\!]_\varepsilon\ x
\end{aligned}
$$

Fig. 5. Parametricity over exceptional translation

3 Kreisel Meets Martin-Löf

It is well-known that Reynolds' parametricity [13] and Kreisel's modified realizability [4] are two instances of the broader logical relation techniques. Usually, parametricity is used to derive theorems for free, while realizability constrains programs. In a surprising turn of events, we use Bernardy's variant of parametricity on CIC [5] as a realizability trick to evict undesirable behaviours of $\mathcal{T}_{\mathbb{E}}$. This leads to the *parametric exceptional translation*, which can be seen as the embodiment of Kreisel's realizability in type theory. In this section, we first present this translation on the negative fragment, then extend it to CIC and finally discuss its meta-theoretical properties.

3.1 Exceptional Parametricity in a Negative World

The exceptional parametricity translation for terms of CC_ω is defined in Fig. 5. Intuitively, any type A in $\mathcal{T}_{\mathbb{E}}$ is turned into a validity predicate $A_\varepsilon : A \to \Box$ which encodes the fact that an inhabitant of A is not allowed to generate unhandled exceptions. For instance, a function is valid if its application to a valid term produces a valid answer. It does not say anything about the application to invalid terms though, which amounts to a *garbage in, garbage out* policy. The translation then states that every pure term is automatically valid.

This translation is exactly standard parametricity for type theory [5] but parametrized by the exceptional translation. This means that any occurrence of a term of the original theory used in the parametricity translation is replaced by its exceptional translation, using $[\cdot]$ or $[\![\cdot]\!]$ depending on whether it is used as a term or as a type. For instance, the translation of an application $[M\ N]_\varepsilon$ is given by $[M]_\varepsilon\ [N]\ [N]_\varepsilon$ instead of just $[M]_\varepsilon\ N\ [N]_\varepsilon$.

Lemma 18 (Substitution lemma). *The translation satisfies the following conversion:* $[M\{x := N\}]_\varepsilon \equiv [M]_\varepsilon\{x := [N], x_\varepsilon := [N]_\varepsilon\}$.

Theorem 19 (Soundness). *The two following properties hold.*

– *If* $M \equiv N$ *then* $[M]_\varepsilon \equiv [N]_\varepsilon$.
– *If* $\Gamma \vdash M : A$ *then* $[\![\Gamma]\!]_\varepsilon \vdash [M]_\varepsilon : [\![A]\!]_\varepsilon\ [M]$.

Proof. By induction on the derivation.

We can use this result to construct another syntactic model of CC_ω. Contrarily to usual syntactic models where sequents are straightforwarldy translated to sequents, this model is slightly more subtle as sequents are translated to pairs of sequents instead. This is similar to the usual parametricity translation.

Definition 20. *The theory* $\mathcal{T}_{\mathbb{E}}^p$ *is defined by the following data.*

– *Terms of* $\mathcal{T}_{\mathbb{E}}^p$ *are pairs of terms of* $\mathcal{T}$.
– *Contexts of* $\mathcal{T}_{\mathbb{E}}^p$ *are pairs of contexts of* $\mathcal{T}$.
– $\vdash_{\mathcal{T}_{\mathbb{E}}^p} \Gamma$ *whenever* $\vdash_{\mathcal{T}} [\![\Gamma]\!]$ *and* $\vdash_{\mathcal{T}} [\![\Gamma]\!]_\varepsilon$.
– $M \equiv_{\mathcal{T}_{\mathbb{E}}^p} N$ *whenever* $[M] \equiv_{\mathcal{T}} [N]$ *and* $[M]_\varepsilon \equiv_{\mathcal{T}} [N]_\varepsilon$.
– $\Gamma \vdash_{\mathcal{T}_{\mathbb{E}}^p} M : A$ *whenever* $[\![\Gamma]\!] \vdash_{\mathcal{T}} [M] : [\![A]\!]$ *and* $[\![\Gamma]\!]_\varepsilon \vdash_{\mathcal{T}} [M]_\varepsilon : [\![A]\!]_\varepsilon\ [M]$.

Once again, Theorem 19 can be rephrased in terms of preservation of theories and syntactic models.

Theorem 21. *If* $\mathcal{T}$ *interprets* CC_ω *then so does* $\mathcal{T}_{\mathbb{E}}^p$. *That is, the parametric exceptional translation is a syntactic model of* CC_ω.

This construction preserves definitional η-expansion, as functions are mapped to (slightly more complicated) functions.

Lemma 22. *If* $\mathcal{T}$ *satisfies definitional* η*-expansion, then so does* $\mathcal{T}_{\mathbb{E}}^p$.

Proof. The first component of the translation preserves definitional η-expansion because functions are mapped to functions. It remains to show that

$$[\lambda x : A.\, M\ x]_\varepsilon := \lambda(x : [\![A]\!])\,(x_\varepsilon : [\![A]\!]_\varepsilon\ x).\, [M]_\varepsilon\ x\ x_\varepsilon \equiv [M]_\varepsilon$$

which holds by applying η-expansion twice.

It is interesting to remark that Bernardy-style unary parametricity also leads to a syntactic model $\mathcal{T}^p$ that interprets CC_ω (as well as CIC), using the same kind of glueing construction. Nonetheless, this model is somewhat degenerate from the logical point of view. Namely it is a conservative extension of the target theory. Indeed, if $\Gamma \vdash_{\mathcal{T}^p} M : A$ for some Γ, M and A from $\mathcal{T}$, then there we also have $\Gamma \vdash_{\mathcal{T}} M : A$, because the first component of the model is the identity, and the original sequent can be retrieved by the first projection.

This is definitely *not* the case with the $\mathcal{T}_{\mathbb{E}}^p$ theory, because the first projection is not the identity. In particular, because of Theorem 16, every sequent in the first projection is inhabited, although it is not the case in $\mathcal{T}$ itself if it is consistent. This means that parametricity can actually bring additional expressivity when it applies to a theory which is not pure, as it is the case here.

$$
\begin{array}{l}
\texttt{Ind bool}_\varepsilon\ : \texttt{bool}^\bullet \to \square := \\
\mid \texttt{true}_\varepsilon : \texttt{bool}_\varepsilon\ \texttt{true}^\bullet \\
\mid \texttt{false}_\varepsilon : \texttt{bool}_\varepsilon\ \texttt{false}^\bullet \\
\texttt{Ind list}_\varepsilon\ (A : \texttt{type})\,(A_\varepsilon : [\![A]\!] \to \square)\ : \texttt{list}^\bullet\ A \to \square := \\
\mid \texttt{nil}_\varepsilon : \texttt{list}_\varepsilon\ A\ A_\varepsilon\ (\texttt{nil}^\bullet\ A) \\
\mid \texttt{cons}_\varepsilon : \Pi(x : [\![A]\!])\,(x_\varepsilon : A_\varepsilon\ x)\,(l : \texttt{list}^\bullet\ A)\,(l_\varepsilon : \texttt{list}_\varepsilon\ A\ A_\varepsilon\ l). \\
\qquad\qquad \texttt{list}_\varepsilon\ A\ A_\varepsilon\ (\texttt{cons}^\bullet\ A\ x\ l) \\
\texttt{Ind eq}_\varepsilon\ (A : \texttt{type})\,(A_\varepsilon : [\![A]\!] \to \square)\,(x : [\![A]\!])\,(x_\varepsilon : A_\varepsilon\ x)\ : \\
\quad \Pi(y : [\![A]\!])\,(y_\varepsilon : A_\varepsilon\ y).\,\texttt{eq}^\bullet\ A\ x\ y \to \square := \\
\mid \texttt{refl}_\varepsilon : \texttt{refl}_\varepsilon\ A\ A_\varepsilon\ x\ x_\varepsilon\ x\ x_\varepsilon\ (\texttt{refl}^\bullet\ A\ x)
\end{array}
$$

Fig. 6. Examples of parametric translation of inductive types

3.2 Exceptional Parametric Translation of CIC

We now describe the parametricity translation of the positive fragment. The intuition is that as it stands for an exception, the default constructor is always invalid, while all other constructors are valid, assuming their arguments are.

Type and Constructor Translation

Definition 23. *Let $\mathcal{I}$ be an inductive type as given in Definition 6. We define the exceptional parametricity translation $\mathcal{I}_\varepsilon$ of $\mathcal{I}$ as the inductive type defined by:*

- *parameters* $[\![p_1 : P_1, \ldots, p_n : P_n]\!]_\varepsilon$;
- *indices* $[\![i_1 : I_1, \ldots, i_m : I_m]\!]_\varepsilon, x : \mathcal{I}\ p_1\ \ldots\ p_n\ i_1\ \ldots\ i_m$;
- *constructors*

$$
\begin{array}{l}
c_{1\varepsilon} : \Pi[\![\vec{a}_1 : \vec{A}_1]\!]_\varepsilon. \\
\qquad \mathcal{I}_\varepsilon\ p_1\ p_{1\varepsilon}\ \cdots\ p_n\ p_{n\varepsilon}\ [V_{1,1}]\ [V_{1,1}]_\varepsilon\ \cdots\ [V_{1,m}]\ [V_{1,m}]_\varepsilon\ (c_1^\bullet\ \vec{p}\ \vec{a}_1) \\
\ldots \\
c_{k\varepsilon} : \Pi[\![\vec{a}_k : \vec{A}_k]\!]_\varepsilon. \\
\qquad \mathcal{I}_\varepsilon\ p_1\ p_{1\varepsilon}\ \cdots\ p_n\ p_{n\varepsilon}\ [V_{k,1}]\ [V_{k,1}]_\varepsilon\ \cdots\ [V_{k,m}]\ [V_{k,m}]_\varepsilon\ (c_k^\bullet\ \vec{p}\ \vec{a}_k).
\end{array}
$$

and we extend the translation as

$$[\mathcal{I}]_\varepsilon := \mathcal{I}_\varepsilon \quad [c_1]_\varepsilon := c_{1\varepsilon} \quad \ldots \quad [c_k]_\varepsilon := c_{k\varepsilon}.$$

Example 24. We give the exceptional parametric inductive translation of our running examples in Fig. 6.

Note that contrarily to the negative case, the exceptional parametricity translation on inductive types is *not* the same thing as the composition of Bernardy's parametricity together with the exceptional translation. Indeed, the latter would also have produced a constructor for the default case from the exceptional inductive translation, whereas our goal is precisely to rule this case out via the additional realizability-like interpretation.

It is also very different from our previous parametric weaning translation [2], which relies on internal parametricity to recover dependent elimination, enforcing by construction that no effectful term exists. Here, effectful terms may be used in the first component, but they are required after the fact to have no inconsistent behaviour. Intuitively, parametric weaning produces one pure sequent, while exceptional parametricity produces two, with the first one being potentially impure and the second one assuring the first one is harmless.

Pattern-Matching Translation

Definition 25. *Let Q be the pattern-matching defined in Definition 12. We pose $[Q]_\varepsilon$ to be the pattern-matching*

$$\begin{aligned}
&\texttt{match}\ [M]_\varepsilon\ \texttt{return}\ \lambda[\![\vec{i} : \vec{I}]\!]_\varepsilon\,(x : \mathcal{I}^\bullet\ [X_1]\ \ldots\ [X_n]\ i_1\ \ldots\ i_m).\\
&\qquad\qquad (x_\varepsilon : \mathcal{I}_\varepsilon\ [X_1]\ [X_1]_\varepsilon\ \ldots\ [X_n]\ [X_n]_\varepsilon\ i_1\ i_{1\varepsilon}\ \ldots\ i_m\ i_{m\varepsilon}\ x)\\
&\qquad\qquad\quad [\![R]\!]_\varepsilon\ [Q_x]\\
&\texttt{with}\\
&\ |\ c_{1\varepsilon}\ a_{1,1}\ a_{1,1\varepsilon}\ \ldots\ a_{1,l_1}\ a_{1,l_1\varepsilon}\ \Rightarrow\ [N_1]_\varepsilon\\
&\ldots\\
&\ |\ c_{k\varepsilon}\ a_{k,1}\ a_{k,1\varepsilon}\ \ldots\ a_{k,l_k}\ a_{1,l_k\varepsilon}\ \Rightarrow\ [N_k]_\varepsilon\\
&\texttt{end}
\end{aligned}$$

where Q_x is the following pattern-matching

$$\begin{aligned}
&\texttt{match}\ x\ \texttt{return}\ \lambda(i_1 : I_1)\ldots(i_m : I_m)\,(x : \mathcal{I}\ X_1\ \ldots\ X_n\ i_1\ \ldots\ i_m).\,R\ \texttt{with}\\
&\ |\ c_1\ a_{1,1}\ \ldots\ a_{1,l_1}\ \Rightarrow\ N_1\\
&\ldots\\
&\ |\ c_k\ a_{k,1}\ \ldots\ a_{k,l_k}\ \Rightarrow\ N_k\\
&\texttt{end}
\end{aligned}$$

that is Q where the scrutinee has been turned into the index variable of the parametricity predicate.

Lemma 26. *With notations and typing assumptions from Definition 12, we have*

$$[\![\Gamma]\!]_\varepsilon \vdash [Q]_\varepsilon : [\![R\{\vec{i} := \vec{Y}, x := M\}]\!]_\varepsilon\ [Q].$$

The exceptional parametricity translation can be extended to handle fixpoints as well, with a few limitations. Translating generic fixpoints uniformly is indeed an open problem in standard parametricity, and our variant faces the same issue. In practice, standard recursors can be automatically translated, and fancy fixpoints may require hand-writing the parametricity proof. We do not describe the recursor translation here though, as it is essentially the same as standard parametricity. Again, the interested reader may test the Coq plugin exposed in Sect. 6 to see how recursors are translated.

Packing everything together allows to state the following result.

Theorem 27. *If $\mathcal{T}$ interprets* CIC*, then so does $\mathcal{T}^p_{\mathbb{E}}$, and thus the exceptional parametricity translation is a syntactic model of* CIC*.*

3.3 Meta-Theoretical Properties of $\mathcal{T}^p_{\mathbb{E}}$

Being built as a syntactic model, $\mathcal{T}^p_{\mathbb{E}}$ inherits a lot of meta-theoretical properties of $\mathcal{T}$. We list a few of interest below.

Theorem 28. *If $\mathcal{T}$ is consistent, then so is $\mathcal{T}^p_{\mathbb{E}}$.*

Proof. Assume $\vdash_{\mathcal{T}^p_{\mathbb{E}}} M_0 : \texttt{empty}$ for some M_0. Then by definition, there exists two terms M and M_ε such that $\vdash_{\mathcal{T}} M : \texttt{empty}^\bullet$ and $\vdash_{\mathcal{T}} M_\varepsilon : \texttt{empty}_\varepsilon\ M$. But $\texttt{empty}_\varepsilon$ has no constructor, and $\mathcal{T}$ is inconsistent.

More generally, the same argument holds for any inductive type.

Theorem 29. *If $\mathcal{T}$ enjoys canonicity, then so does $\mathcal{T}^p_{\mathbb{E}}$.*

Proof. The exceptional parametricity translation for inductive types has the same structure as the original type, so any normal form in $\mathcal{T}^p_{\mathbb{E}}$ can be mapped back to a normal form in $\mathcal{T}$.

4 Effectively Extending CIC

The parametric exceptional translation allows to extend the logical expressivity of CIC in the following ways, which we develop in the remainder of this section.

We show in Sect. 4.1 that Markov's rule is admissible in CIC. We already sketched this result in our previous paper [2], but we come back to it in more details. More generally, we show a form of conservativity of double-negation elimination over the type-theoretic version of Π^0_2 formulae.

In Sect. 4.2, we exhibit a syntactic model of CIC which satisfies definitional η-expansion for functions but which negates function extensionality. As far as we know, this was not known.

Finally, in Sect. 4.3, we show that there exists a model of CIC which validates the independence of premises. This is a new result, that shows that CIC can feature traces of classical reasoning while staying computational. We use this result in Sect. 4.4 to give an alternative proof of the recent result of Coquand and Mannaa [7] that Markov's principle is not provable in CIC.

4.1 Markov's Rule

We show in this section that CIC is closed under a generalized Markov's rule. The technique used here is no more than a dependently-typed variant of Friedman's trick [14]. Indeed, Friedman's A-translation amounts to add exceptions to intuitionistic logic, which is precisely what $\mathcal{T}_{\mathbb{E}}$ does for CIC.

Definition 30. *An inductive type in* CIC *is said to be first-order if all the types of the arguments of its constructors, in its parameters and in its indices are recursively first-order.*

Example 31. The $\texttt{empty}$, $\texttt{unit}$ and $\mathbb{N}$ types are first-order. If P and Q are first-order then so is $\Sigma p : P.\, Q$, $P + Q$ and $\texttt{eq}\ P\ p_0\ p_1$. Consequently, the CIC equivalent of Σ_1^0 formulae are in particular first-order.

First-order types enjoy uncommon properties, like the fact that they can be injected into effectful terms and purified away. This is then used to prove the generalized Markov's Rule.

Lemma 32. *For every first-order type $\vec{p} : \vec{P} \vdash Q : \square$ where all $\vec{P}$ are first-order, there are retractions $\iota_{\vec{P}}$, ι_Q and $\theta_{\vec{P}}$, θ_Q s.t.:*

$$\vec{p} : \vec{P} \vdash \iota_Q : Q \to [\![Q]\!]\{\vec{p} := \iota_{\vec{P}}\ \vec{p}\}$$
$$\vec{p} : \vec{P} \vdash \theta_Q : [\![Q]\!]\{\vec{p} := \iota_{\vec{P}}\ \vec{p}\} \to Q + \mathbb{E}.$$

Proof. The ι terms exist because effectful inductive types are a semantical superset of their pure equivalent, and the θ terms are implemented by recursively forcing the corresponding impure inductive term. One relies on decidability of equality of first-order type to fix the indices.

Theorem 33 (Generalized Markov's Rule). *For any first-order type P and first-order predicate Q over P, if $\vdash_{\mathrm{CIC}} \Pi p : P. \neg\neg\ (Q\ p)$ then $\vdash_{\mathrm{CIC}} \Pi p : P.\, Q\ p$.*

Proof. Let $\vdash M : \Pi p : P. \neg\neg\ (Q\ p)$. By taking $\mathbb{E} := Q\ p$ and apply the soundness theorem, one gets a proof

$$p : P \vdash [M] : \Pi \hat{p} : [\![P]\!].\, ([\![Q\ \hat{p}]\!] \to \texttt{empty}^\bullet) \to \texttt{empty}^\bullet.$$

But $\texttt{empty}^\bullet \cong \mathbb{E} \equiv Q\ p$, so we can derive from $[M]$ a term $M^\sharp$ s.t.

$$p : P \vdash M^\sharp : \Pi \hat{p} : [\![P]\!].\, ([\![Q\ \hat{p}]\!] \to Q\ p + Q\ p) \to Q\ p.$$

The proofterm we were looking for is thus no more than $\lambda p : P.\, M^\sharp\ (\iota_P\ p)\ \theta_Q$.

4.2 Function Intensionality with η-expansion

In a previous paper [9], we already showed that there existed a syntactic model of CIC that allowed to internally disprove function extensionality. Yet, this model was clearly not preserving definitional η-expansion on functions, as it was adding additional structure to abstraction and application (namely a boolean). Thanks to our new model, we can now demonstrate that counterintuitively, it is possible to have a consistent type theory that enjoys definitional η-expansion while negating internally function extensionality. In this section we suppose that $\mathbb{E} := \texttt{unit}$, although any inhabited type of exceptions would work.

By Lemma 22, we know that the parametric exceptional translation preserves definitional η-expansion. It is thus sufficient to find two functions that are extensionally equal but intensionally distinct in the model. Let us consider to this end the $\texttt{unit} \to \texttt{unit}$ functions

$$\texttt{id}_\bot := \lambda u : \texttt{unit}.\, u \qquad\qquad \texttt{id}_\top := \lambda u : \texttt{unit}.\, \texttt{tt}.$$

Theorem 34. *The following sequents are derivable:*

$$\vdash_{\mathcal{T}_{\mathbb{E}}^{p}} \Pi u : \mathtt{unit}.\, \mathtt{id}_{\bot}\ u = \mathtt{id}_{\top}\ u \qquad \vdash_{\mathcal{T}_{\mathbb{E}}^{p}} \mathtt{id}_{\bot} = \mathtt{id}_{\top} \to \mathtt{empty}.$$

Proof. The main difference between the two functions is that $\mathtt{id}_{\bot}$ preserves exceptions while $\mathtt{id}_{\top}$ does not, which we exploit.

The first sequent is provable in CIC by dependent elimination and thus is derivable in $\mathcal{T}_{\mathbb{E}}^{p}$ by applying the soundness theorem.

To prove the first component of the second sequent, we exhibit a property that discriminates $[\mathtt{id}_{\bot}]$ and $[\mathtt{id}_{\top}]$, which is, as explained, their evaluation on the term $\mathtt{unit}_{\varnothing}\ \mathtt{tt}$. Showing then that this proof is parametric is equivalent to showing $\Pi(p : [\![\mathtt{id}_{\bot} = \mathtt{id}_{\top}]\!])\,(p_{\varepsilon} : [\![\mathtt{id}_{\bot} = \mathtt{id}_{\top}]\!]_{\varepsilon}\ p).\, \mathtt{empty}$. But p_{ε} actually implies $[\mathtt{id}_{\bot}] = [\mathtt{id}_{\top}]$, which we just showed was absurd.

4.3 Independence of Premise

Independence of premise (IP) is a semi-classical principle from first-order logic whose CIC equivalent can be stated as follows.

$$\Pi(A : \square)\,(B : \mathbb{N} \to \square).\, (\neg A \to \Sigma n : \mathbb{N}.\, B\ n) \to \Sigma n : \mathbb{N}.\, \neg A \to B\ n \quad \text{(IP)}$$

Although not derivable in intuitionistic logic, it is an admissible rule of **HA**. The standard proof of this property is to go through Kreisel's modified realizability interpretation of **HA** [4]. In a nutshell, the interpretation goes as follows: by induction over a formula A, define a simple type $\tau(A)$ of realizers of A together with a realizability predicate $\cdot \Vdash A$ over $\tau(A)$. Then show that whenever $\vdash_{\mathbf{HA}} A$, there exists some simply-typed term $t : \tau(A)$ s.t. $t \Vdash A$. As the interpretation also implies that there is no t s.t. $t \Vdash \bot$, this gives a sound model of **HA**, which contains more than the latter. Most notably, there is for instance a term $\mathtt{ip}$ s.t.

$$\mathtt{ip} \Vdash (\neg A \to \exists n.\, B) \to \exists n.\, \neg A \to B$$

for any A, B. Intriguingly, the computational content of $\mathtt{ip}$ did not seem to receive a fair treatment in the literature. To the best of our knowledge, it has never been explicitly stated that IP was realizable because of the following "bug" of Kreisel's modified realizability.

Lemma 35 (Kreisel's bug). *For every formula A, $\tau(A)$ is inhabited. In particular, $\tau(\bot) := \mathtt{unit}$.*

We show that this is actually not a bug, but a hidden feature of Kreisel's modified realizability, which secretly allows to encode exceptions in the realizers. To this end, we implement IP in $\mathcal{T}_{\mathbb{E}}^{p}$ by relying internally on *paraproofs*, i.e. terms raising exceptions, while ensuring these exceptions never escape outside of the locally unsafe boundary. The resulting $\mathcal{T}_{\mathbb{E}}^{p}$ term has essentially the same computational content as its Kreisel's realizability counterpart. In this section we suppose that $\mathbb{E} := \mathtt{unit}$, although assuming $\mathbb{E}$ to be inhabited is sufficient.

To ease the understanding of the definition, we rely on effectful combinators that can be defined in $\mathcal{T}_{\mathbb{E}}$.

Definition 36. *We define in $\mathcal{T}_{\mathbb{E}}$ the following terms.*

$$\begin{aligned}
&\texttt{fail} \;:\; \Pi A : \square.\, A\\
&[\texttt{fail}] := \lambda A : [\![\square]\!].\, [A]_{\varnothing}\ \texttt{tt}
\end{aligned}$$

$$\begin{aligned}
&\texttt{is}_\Sigma \;:\; \Pi A\, B.\, (\Sigma x : A.\, B) \to \texttt{bool}\\
&[\texttt{is}_\Sigma] := \lambda A\, B\, p.\ \texttt{match}\ p\ \texttt{with}\\
&\qquad \mid \texttt{ex}^\bullet\ _\ _ \Rightarrow \texttt{true}^\bullet\\
&\qquad \mid \Sigma_{\varnothing}\ _ \Rightarrow \texttt{false}^\bullet\\
&\qquad \texttt{end}
\end{aligned}$$

$$\begin{aligned}
&\texttt{is}_{\mathbb{N}} \;:\; \mathbb{N} \to \texttt{bool}\\
&[\texttt{is}_{\mathbb{N}}] := \texttt{fix}\ \texttt{is}_{\mathbb{N}}\ n\ :=\ \texttt{match}\ n\ \texttt{with}\\
&\qquad \mid \texttt{O}^\bullet \Rightarrow \texttt{true}^\bullet\\
&\qquad \mid \texttt{S}^\bullet\ n \Rightarrow \texttt{is}_{\mathbb{N}}\ n\\
&\qquad \mid \mathbb{N}_{\varnothing}\ _ \Rightarrow \texttt{false}^\bullet\\
&\qquad \texttt{end}
\end{aligned}$$

It is worth insisting that these combinators are not necessarily parametric. While it can be shown that $\texttt{is}_\Sigma$ and $\texttt{is}_{\mathbb{N}}$ actually are, `fail` is luckily not. The $\texttt{is}_\Sigma$ and $\texttt{is}_{\mathbb{N}}$ functions are used in order to check that a value is actually pure and does not contain exceptions.

Definition 37. *We define* `ip` *in $\mathcal{T}_{\mathbb{E}}$ in direct style below, using the available combinators from Definition 36 and a bit of syntactic sugar.*

$$\begin{aligned}
&\texttt{ip} :\ \mathrm{IP}\\
&\texttt{ip} := \lambda(A : \square)\,(B : \mathbb{N} \to \square)\,(f : \neg A \to \Sigma n : \mathbb{N}.\, B\ n).\\
&\qquad \texttt{let}\ p\ :=\ f\ (\texttt{fail}\ (\neg A))\ \texttt{in}\\
&\qquad \texttt{if}\ \texttt{is}_\Sigma\ \mathbb{N}\ B\ p\ \texttt{then match}\ p\ \texttt{with}\\
&\qquad \mid\ \texttt{ex}\ n\ b\ \Rightarrow\ \texttt{if}\ \texttt{is}_{\mathbb{N}}\ n\ \texttt{then ex}\ _\ _\ n\ (\lambda_ : \neg A.\, b)\\
&\qquad\qquad\qquad\qquad\qquad \texttt{else ex}\ _\ _\ \texttt{O}\ (\texttt{fail}\ (\neg A \to B\ \texttt{O}))\\
&\qquad \texttt{end else ex}\ _\ _\ \texttt{O}\ (\texttt{fail}\ (\neg A \to B\ \texttt{O}))
\end{aligned}$$

The intuition behind this term is the following. Given $f : \neg A \to \Sigma n : \mathbb{N}.\, B\ n$, we apply it to a dummy function which fails whenever it is used. Owing to the semantics of negation, we know *in the parametricity layer* that the only way for this application to return an exception is that f actually contained a proof of A and applied `fail` to it. Therefore, given a true proof of $\neg A$, we are in an inconsistent setting and thus we are able to do whatever pleases us. The issue is that we do not have access to such a proof yet, and we do have to provide a valid integer now. Therefore, we check whether f actually provided us with a valid pair containing a valid integer. If so, this is our answer, otherwise we stuff a dummy integer value and we postpone the contradiction.

This is essentially the same realizer as the one from Kreisel's modified realizability, except that we have a fancy type system for realizers. In particular, because we have dependent types, integers also exist in the logical layer, so that they need to be checked for exceptions as well. The only thing that remains to be proved is that `ip` also lives in $\mathcal{T}^p_{\mathbb{E}}$.

Theorem 38. *There is a proof of $\vdash_{\mathcal{T}} [\![\mathrm{IP}]\!]_\varepsilon\ [\texttt{ip}]$.*

Proof. The proof is straightforward but tedious, so we do not give the full details. The file `IPc.v` of the companion Coq plugin contains an explicit proof. The essential properties that make it go through are the following.

- $\vdash_{\mathcal{T}} \Pi(n : \mathbb{N}^\bullet)\,(p_1\,p_2 : \mathbb{N}_\varepsilon\ n).\, p_1 = p_2$
- $\vdash_{\mathcal{T}} \Pi n : \mathbb{N}^\bullet.\, [\mathtt{is}_{\mathbb{N}}]\ n = \mathtt{true}^\bullet \leftrightarrow \mathbb{N}_\varepsilon\ n$
- $\vdash_{\mathcal{T}} \Pi(p\,q : [\![\neg A]\!]).\, [\![\neg A]\!]_\varepsilon\ p \to [\![\neg A]\!]_\varepsilon\ q$

Corollary 39. *We have* $\vdash_{\mathcal{T}^p_{\mathbb{E}}}$ IP.

4.4 Non-provability of Markov's Principle

From this result, one can get a very easy syntactic proof of the independence result of Markov's principle from CIC. Markov's principle is usually stated as

$$\Pi P : \mathbb{N} \to \mathtt{bool}.\, \neg\neg\, (\Sigma n : \mathbb{N}.\, P\ n = \mathtt{true}) \to \Sigma n : \mathbb{N}.\, P\ n = \mathtt{true} \quad \text{(MP)}$$

An independence result was recently proved by Coquand and Mannaa by a semantic argument [7]. We leverage instead a property from realizability [15] that has been applied to type theory the other way around by Herbelin [16].

Lemma 40. *If* $\mathcal{S}$ *is a computable theory containing* CIC *and enjoying canonicity, then one cannot have both* $\vdash_{\mathcal{S}}$ IP *and* $\vdash_{\mathcal{S}}$ MP.

Proof. By applying IP to MP, one easily obtains that

$$\vdash_{\mathcal{S}} \Pi P : \mathbb{N} \to \mathtt{bool}.\, \Sigma n : \mathbb{N}.\, \Pi m : \mathbb{N}.\, P\ m = \mathtt{true} \to P\ n = \mathtt{true}.$$

Thus, for every closed $P : \mathbb{N} \to \mathtt{bool}$, by canonicity there exists a closed $n_P : \mathbb{N}$ s.t. $\vdash_{\mathcal{S}} \Pi m : \mathbb{N}.\, P\ m = \mathtt{true} \to P\ n_P = \mathtt{true}$. But then one can decide whether P holds for some n by just computing $P\ n_P$, so that we effectively obtained an oracle deciding the halting problem (which is expressible in CIC).

Corollary 41. *We have* $\nvdash_{\mathrm{CIC}^p_{\mathbb{E}}}$ MP *and thus also* $\nvdash_{\mathrm{CIC}}$ MP.

5 Possible Extensions

5.1 Negative Records

Interestingly, the fact that the translation introduces effects has unintented consequences on a few properties of type theory that are often taken for granted. Namely, because type theory is pure, there is a widespread confusion amongst type theorists between positive tuples and negative records.

- Positive tuples are defined as a one-constructor inductive type, introduced by this constructor and eliminated by pattern-matching. They do not (and in general cannot, for typing reasons) satisfy definitional η-laws, also known as *surjective pairing*.
- Negative records are defined as a record type, introduced by primitive packing and eliminated by projections. They naturally obey definitional η-laws.

$$A, B, M, N ::= \ldots \mid \&x : A.\, B \mid \langle M, N \rangle \mid M.\pi_1 \mid M.\pi_2$$

$$\frac{\Gamma \vdash A : \square_i \quad \Gamma, x : A \vdash B : \square_j}{\Gamma \vdash \&x : A.\, B : \square_{\max(i,j)}} \qquad \frac{\Gamma \vdash M : \&x : A.\, B}{\Gamma \vdash M.\pi_1 : A} \qquad \frac{\Gamma \vdash M : \&x : A.\, B}{\Gamma \vdash M.\pi_2 : B\{x := M.\pi_1\}}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma, x : A \vdash B : \square \qquad \Gamma \vdash N : B\{x := M\}}{\Gamma \vdash \langle M, N \rangle : \&x : A.\, B}$$

$$\langle M.\pi_1, M.\pi_2 \rangle \equiv M \qquad \langle M, N \rangle.\pi_1 \equiv M \qquad \langle M, N \rangle.\pi_2 \equiv N$$

Fig. 7. Negative pairs

$$\begin{aligned}
[\&x : A.\, B] &:= \texttt{TypeVal}\ (\&x : [\![A]\!].\, [\![B]\!])\ (\lambda e : \mathbb{E}.\, \langle [A]_{\varnothing}\ e, [B]_{\varnothing}\{x := [A]_{\varnothing}\ e\}\ e \rangle) \\
[\langle M, N \rangle] &:= \langle [M], [N] \rangle \\
[M.\pi_i] &:= [M].\pi_i
\end{aligned}$$

Fig. 8. Exceptional translation of negative pairs

In the remainder of this section, we will focus on the specific case of pairs, but the same arguments are generalizable to arbitrary records. Positive pairs $\Sigma x : A.\, B$ are defined by the inductive type from Fig. 4. Negative pairs $\&x : A.\, B$ are defined as a primitive structure in Fig. 7. We use the ampersand notation as a reference to linear logic.

In CIC, it is possible to show that negative and positive pairs are propositionally isomorphic, because positive pairs enjoy dependent elimination. Nonetheless, it is a well-known fact in the programming folklore that in a call-by-name language with effects, the two are sharply distinct. For instance, in presence of exceptions, assuming $\vdash M : \Sigma x : A.\, B$, one does not have in general

$$M \equiv \texttt{ex}\ A\ B\ (\texttt{fst}\ A\ B\ M)\ (\texttt{snd}\ A\ B\ M)$$

where `fst` and `snd` are defined by pattern-matching. Indeed, if M is itself an exception, the two sides can be discriminated by a pattern-matching. Matching on the left-hand side results in immediate reraising of the exception, while matching on the right-hand side succeeds as long as the arguments of the constructor are not forced. Forcefully equating those two terms would then result in a trivial equational theory.

Such a phenomenon is at work in the exceptional translation. It is actually possible to interpret negative pairs through the translation, but in a way that significantly differs from the translation of positive pairs. In this section, we assume that $\mathcal{T}$ contains negative pairs.

Definition 42. *The translation of negative pairs is given in Fig. 8.*

It is straightforward to check that the definitions of Fig. 8 preserve the conversion and typing rules from Fig. 7. The same translation can be extended to any record. We thus have the following theorem.

Theorem 43. *If $\mathcal{T}$ has negative records, then so has $\mathcal{T}_{\mathbb{E}}$.*

It is enlightening to look at the difference between negative and positive pairs through the translation, because now we have effects that allow to separate them clearly. Indeed, compare

$$[\![\&x : A.\, B]\!] \equiv \&x : [\![A]\!].\, [\![B]\!] \quad \text{with} \quad [\![\Sigma x : A.\, B]\!] \cong \mathbb{E} + \Sigma x : [\![A]\!].\, [\![B]\!].$$

Clearly, if $\mathbb{E}$ is inhabited, then the two types do not even have the same cardinal, assuming A and B are finite. Furthermore, their default inhabitant is not the same at all. It is defined pointwise for negative pairs, while it is a special constructor for positive ones. Finally, there is obviously not any chance that $[\![\Sigma x : A.\, B]\!]$ satisfies definitional surjective pairing in vanilla CIC, as it has two constructors. The trick is that the two types are externally distinguishable, but are not internally so, because $\mathcal{T}_{\mathbb{E}}$ is a model of CIC+& and thus proves that they are propositionally isomorphic.

It is possible to equip negative pairs with a parametricity relation defined as a primitive record which is the pointwise parametricity relation of each field, which naturally preserve typing and conversion rules.

Theorem 44. *If $\mathcal{T}$ has negative records, then so has $\mathcal{T}^p_{\mathbb{E}}$.*

5.2 Impredicative Universe

All the systems we have considered so far are predicative. It is nonetheless possible to implement an impredicative universe $*$ in $\mathcal{T}_{\mathbb{E}}$ if $\mathcal{T}$ features one.

Intuitively, it is sufficient to ask for an inductive type `prop` living in $\square_i$ for all i, which is defined just as `type`, except that its constructor `PropVal` corresponding to `TypeVal` contains elements of $*$ rather than $\square$. Then one can similarly define $\texttt{El}_*$ and $\texttt{Err}_*$ acting on `prop` rather than `type`. One then slightly tweaks the $[\![\cdot]\!]$ macro from Fig. 2 by defining it instead as

$$[\![A]\!] := \begin{cases} \texttt{El}_* \ [A] \text{ if } A : * \\ \texttt{El} \ [A] \quad \text{otherwise} \end{cases}$$

and similarly for type constructors. With this modified translation, one obtains a soundness theorem for CC_ω.

Theorem 45. *The exceptional translation is a syntactic model of* $\mathrm{CC}_\omega + *$.

Likewise, the inductive translation is amenable to interpret an impredicative universe, with one major restriction though.

Theorem 46. *The exceptional translation is a syntactic model of* CIC$+*$ *without the singleton elimination rule.*

Indeed, the addition of the default constructor disrupts the singleton elimination criterion for all inductive types. Actually, this criterion is very fragile, and even if $\mathcal{T}_{\mathbb{E}}$ satisfied it, Keller and Lasson showed that the parametricity translation could not interpret inductive types in $*$ for similar reasons [17], and $\mathcal{T}_{\mathbb{E}}^{p}$ would face the same issue.

6 The Exceptional Translation in Practice

6.1 Implementation as a Coq Plugin

The (parametric) exceptional translation is a translation of CIC into itself, which means that we can directly implement it as a Coq plugin. This way, we can use the translation to extend safely Coq with new logical principles, so that typechecking remains decidable.

Such a Coq plugin is simply a program that, given a Coq proof term M, produces the translations $[M]$ and $[M]_{\varepsilon}$ as Coq terms. For instance, the translations of type `list`, given in Figs. 4 and 6, are obtained by typing the following commands, which define each one new inductive type in Coq.

```
Effect Translate list.
Parametricity Translate list.
```

The first command produces only [`list`], while the second produces $[\texttt{list}]_{\varepsilon}$. But the main interest of the translation is that we can exhibit new constructors. For instance, the `raise` operation described in Sect. 2.4 is defined as

```
Effect Definition Exception : Type := fun E ⇒ TypeVal E E id.
Effect Definition raise : ∀ A, Exception → A := fun E (A : type E) ⇒ Err A.
```

6.2 Usecase: A Cast Framework

We can use the ability to raise exception to define partial function in the exceptional layer. For instance, given a decidable property (described by the type class below), it is then possible to define a cast function from `A` to Σ (`a` : `A`). `P a` returning the converted value if the property is satisfied and raising an exception otherwise (using an inhabitant `cast_failed` of `Exception`).

```
Class Decidable (A : Type) := dec : A + (not A).
Definition cast A (P : A → Type) (a:A) {Hdec : Decidable (P a)} : Σ (a : A). P a
:= match dec (P a) with
   | inl p ⇒ (a ; p)
   | inr _ ⇒ raise cast_failed
   end.
```

Using this cast mechanism, it is easy to define a function `list_to_pair` from lists to pairs by first converting the list into a list size two, using the impure function `cast` (`list A`) (`fun l ⇒ List.length l = 2`) and then recovering a pair from a list of size two using a pure function.

In the exceptional layer, it is possible to prove the following property

```
Definition list_to_pair_prop A (x y : A) : list_to_pair [x ; y] = (x,y).
```

in at least two way. One can perfectly prove it by simply raising an exception at top level, or by reflexivity—using the fact that `list_to_pair [x ; y]` actually reduces to `(x,y)`.

However, there is a way to distinguish between those two proofs in the target theory, here Coq, by stating the following lemma which can only proven for the proof not raising an exception.

```
Definition list_to_pair_prop_soundness A x y :
    list_to_pair_prop• A x y = eq_refl• _ _ _ := eq_refl _.
```

where underscores represent arguments inferred by Coq.

7 Related Work

Adding Dependency to an Effectful Language. There are numerous works on adding dependent types in mainstream effectful programming languages. They all mostly focused on how to appropriately restrict effectful terms from appearing in types. Indeed, if types only depend on pure terms, the problem of having two different evaluations of the effect of the term (at the level of types and at the level of terms) disappear. This is the case for instance for Dependent ML of Xi and Pfenning [18], or more recently for Casinghino *et al.* [19] on how to combine proofs and programs when programs can be non-terminating. The $F^\star$ programming language of Swamy *et al.* [20] uses a notion of primitive effects including state, exceptions, divergence and IO. Each effect is described through a monadic predicate transformer semantics which allows to have a pure core dependent language to reason on those effects. On a more foundational side, there are two recent and overlapping lines of work on the description of a dependent call-by-push-value (CBPV) by Ahman *et al.* [21] and Vákár [22]. Those works also use a purity restriction for dependency, but using the CBPV language, deals with any effect described in monadic style. On another line of work, Brady advocates for the use of algebraic effects as an elegant way to allow combing effects more smoothly than with a monadic approach and gives an implementation in Idris [23].

Adding Effects to a Dependently-Typed Language. Nanevski *et al.* [24] have developed Hoare type theory (HTT) to extend Coq with monadic style effects. To this end, they provide an axiomatic extension of Coq with a monad in which to encapsulate imperative code. Important tools have been developed on HTT, most notably the Ynot project [25]. Apart from being axiomatic, their monadic approach does not allow to mix effectful programs and dependency but is rather made for proving inside Coq properties on simply typed imperative programs.

Internal Translation of Type Theory. A non-axiomatic way to extend type theory with new features is to use internal translation, that is translation of type theory into itself as advocated by Boulier *et al.* [9]. The presentation of parametricity

for type theory given by Bernardy and Lasson [5] can be seen as one of the first internal translation of type theory. However, this one does not add any new power to type theory as it is a conservative extension. Barthe *et al.* [26] have described a CPS translation for CC_ω featuring `call-cc`, but without dealing with inductive types and relying on a form of type stratification. A variant of this translation has been extended recently by Bowman *et al.* [27] to dependent sums using answer-type polymorphism $\Pi\alpha : \Box.\, (A \to \alpha) \to \alpha$. A generic class of internal translations has been defined by Jaber *et al.* [28] using forcing, which can be seen as a type theoretic version of the presheaf construction used in categorical logic. This class of translation works on all CIC but for a restricted version of dependent elimination, identical to the Baclofen type theory [2]. Therefore, to the best of our knowledge, the exceptional translation is the first complete internal translation of CIC adding a particular notion of effect.

8 Conclusion and Future Work

In this paper, we have defined the exceptional translation, the first syntactic translation of the Calculus of Inductive Constructions into itself, adding effects and that covers full dependent elimination. This results in a new type theory, which features call-by-name exceptions with decidable type-checking and a weaker form of canonicity. We have shown that although the resulting theory is inconsistent, it is possible to reason on exceptional programs and show that some of them actually never raise an exception by relying on the target theory. This provides a sound logical framework allowing to transparently prove safety properties about impure dependently-typed programs. Then, using parametricity, we have given an additional layer at the top of the exceptional translation in order to tame exceptions and preserve consistency. This way, we have consistently extended the logical expressivity of CIC with independence of premises, Markov's rule, and the negation of function extensionality while retaining η-expansion. Both translations have been implemented in a Coq plugin, which we use to formalize the examples.

One of the main directions of future work is to investigate whether other kind of effects can give rise to an internal translation of CIC. To that end, it seems promising to look at algebraic presentation of effects. Indeed, the recent work on the non-necessity of the value restriction policy for algebraic effects and handlers of Kammar and Pretnar [29] suggests that we should be able to perform similar translations on CIC with full dependent elimination for other algebraic effects and handlers than exceptions.

Acknowledgements. This research was supported in part by an ERC Consolidator Grant for the project "RustBelt", funded under Horizon 2020 grant agreement № 683289 and an ERC Starting Grant for the project "CoqHoTT", funded under Horizon 2020 grant agreement № 637339.

References

1. Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1), 55–92 (1991)
2. Pédrot, P., Tabareau, N.: An effectful way to eliminate addiction to dependence. In: 32nd Annual Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, 20–23 June 2017, pp. 1–12 (2017)
3. Munch-Maccagnoni, G.: Models of a non-associative composition. In: Muscholl, A. (ed.) FoSSaCS 2014. LNCS, vol. 8412, pp. 396–410. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54830-7_26
4. Kreisel, G.: Interpretation of analysis by means of constructive functionals of finite types. In: Heyting, A. (ed.) Constructivity in Mathematics, pp. 101–128. North-Holland Pub. Co., Amsterdam (1959)
5. Bernardy, J.-P., Lasson, M.: Realizability and parametricity in pure type systems. In: Hofmann, M. (ed.) FoSSaCS 2011. LNCS, vol. 6604, pp. 108–122. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19805-2_8
6. Avigad, J., Feferman, S.: Gödel's functional ("Dialectica") interpretation. In: The Handbook of Proof Theory, pp. 337–405. North-Holland (1999)
7. Coquand, T., Mannaa, B.: The independence of Markov's principle in type theory. In: 1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, Porto, Portugal, 22–26 June 2016, pp. 17:1–17:18 (2016)
8. Coquand, T., Huet, G.P.: The calculus of constructions. Inf. Comput. **76**(2/3), 95–120 (1988)
9. Boulier, S., Pédrot, P., Tabareau, N.: The next 700 syntactical models of type theory. In: Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, 16–17 January 2017, pp. 182–194 (2017)
10. The Coq Development Team: The Coq proof assistant reference manual (2017)
11. Werner, B.: Une Théorie des Constructions Inductives. Ph.D. thesis, Université Paris-Diderot - Paris VII, May 1994
12. Tanter, É., Tabareau, N.: Gradual certified programming in Coq. In: Proceedings of the 11th ACM Dynamic Languages Symposium (DLS 2015), Pittsburgh, PA, USA, pp. 26–40. ACM Press, October 2015
13. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: IFIP Congress, pp. 513–523 (1983)
14. Friedman, H.: Classically and intuitionistically provably recursive functions. In: Miiller, G.H., Scott, D.S. (eds.) Higher Set Theory. Lecture Notes in Mathematics, pp. 21–27. Springer, Heidelberg (1978)
15. Troelstra, A. (ed.): Metamathematical Investigation of Intuitionistic Arithmetic and Analysis. Lecture Notes in Mathematics, vol. 344. Springer, Heidelberg (1973). https://doi.org/10.1007/BFb0066739
16. Herbelin, H.: An intuitionistic logic that proves Markov's principle. In: Proceedings of the 25th Annual Symposium on Logic in Computer Science, LICS 2010, Edinburgh, United Kingdom, 11–14 July 2010, pp. 50–56 (2010)
17. Keller, C., Lasson, M.: Parametricity in an impredicative sort. In: Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, Fontainebleau, France, 3–6 September 2012, pp. 381–395 (2012)
18. Xi, H., Pfenning, F.: Dependent types in practical programming. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1999, pp. 214–227. ACM, New York (1999)

19. Casinghino, C., Sjöberg, V., Weirich, S.: Combining proofs and programs in a dependently typed language. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, pp. 33–45. ACM, New York (2014)
20. Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoue, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in F*. In: 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 256–270. ACM, January 2016
21. Ahman, D., Ghani, N., Plotkin, G.D.: Dependent types and fibred computational effects. In: Jacobs, B., Löding, C. (eds.) FoSSaCS 2016. LNCS, vol. 9634, pp. 36–54. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49630-5_3
22. Vákár, M.: A framework for dependent types and effects (2015) draft
23. Brady, E.: Idris, a general-purpose dependently typed programming language: design and implementation. J. Funct. Program. **23**(05), 552–593 (2013)
24. Nanevski, A., Morrisett, G., Birkedal, L.: Hoare type theory, polymorphism and separation. J. Funct. Program. **18**(5–6), 865–911 (2008)
25. Chlipala, A., Malecha, G., Morrisett, G., Shinnar, A., Wisnesky, R.: Effective interactive proofs for higher-order imperative programs. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, ICFP 2009, pp. 79–90. ACM, New York (2009)
26. Barthe, G., Hatcliff, J., Sørensen, M.H.B.: CPS translations and applications: the cube and beyond. High. Order Symbol. Comput. **12**(2), 125–170 (1999)
27. Bowman, W., Cong, Y., Rioux, N., Ahmed, A.: Type-preserving CPS translation of σ and π types is not possible. In: Proceedings of the 45th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2018. ACM, New York (2018)
28. Jaber, G., Lewertowski, G., Pédrot, P., Sozeau, M., Tabareau, N.: The definitional side of the forcing. In: Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2016, New York, NY, USA, 5–8 July 2016, pp. 367–376 (2016)
29. Kammar, O., Pretnar, M.: No value restriction is needed for algebraic effects and handlers. J. Funct. Program. **27**, 367–376 (2017)