Internship Report – M2 Informatique Fondamentale
École Normale Supérieure de Lyon

# Gradualizing the Calculus of Inductive Constructions

Meven Bertrand

Under the Supervision of Nicolas Tabareau
Équipe Gallinette, Laboratoire des Sciences du Numérique de Nantes

In Collaboration with Éric Tanter
University of Chile & Inria Paris

August 30, 2019

ENS DE LYON

LS2N LABORATOIRE DES SCIENCES DU NUMÉRIQUE DE NANTES

*I would like to thank those that made these last six months so valuable to me.*

*Nicolas and Éric, for guiding me all the way to this report. I was not really sure of where I was going, but thanks to you I ended up with a result I am proud of.*

*The whole Gallinette team, for making me feel at home from day one and teaching me so much. I hope we have many more passionate conversations, be it on obscure points of type theory or on the CDG Express.*

*Daniel, Marine, Valentin, Joanne, and all the friends and family that have been there every time I needed. I wouldn't have gone through these last months without collapsing, hadn't you been there.*

*Tangi, for the good time we managed to share despite everything, and for the incredible lessons I've learned from you. Farewell.*

# Contents



# Report Overview

GRADUAL TYPING Gradual typing is a kind of typing discipline that aims for a cohabitation of static and dynamic types in one and the same type system, with the possibility for the programmer to control to which extent each piece of code is dynamically or statically typed. The aim is to combine in one system the advantages of both worlds: the static parts come with the usual strong guarantees and efficiency of static types, while the dynamic parts have the flexibility of dynamically typed languages. The most important feature of gradual typing is that this cohabitation is disciplined: a gradually typed language does not just consist of a juxtaposition of a dynamically and a statically typed language. Instead, it gives guarantees on the behaviour of a program when evolving its types alongside the dynamic-static axis, allowing for a smooth transition.

There are two ways to look at gradual typing: one can see it as adding an optional typing discipline to dynamically typed languages, or adding a dynamic component to a static type system. The first alternative appears in practice, for instance in Python starting at version 3.5 [RLL14]. However, from the point of view of theory, as studied type systems are mainly static, the usual challenge is to turn one's favourite (static) type system gradual. This is the approach we want to follow: gradualizing the Calculus of Inductive Constructions, which is most definitely our favourite type system.

THE CALCULUS OF INDUCTIVE CONSTRUCTIONS The Calculus of Inductive Constructions (CIC) is the logical and type-theoretic formalism behind a whole set of proof assistants, notably Coq [The19]. Following the Curry-Howard isomorphism, it can be considered both as a higher-order logic, and as a type system for programs. It is in a way the pinnacle of this isomorphism, insofar as it is very close to the limit of how powerful and

expressive a type system can get without becoming inconsistent – and thus worthless – as a logic. Of course, this comes with a cost, and programming with CIC is quite complex: the more expressive types get, the harder it is to make programs type-check...

GRADUALIZING CIC Although there had already been efforts into applying the gradual approach to type systems close to CIC, at the starting time of the internship, a gradual counterpart to the whole CIC had not yet been achieved. Indeed, CIC presents many challenges to gradualization: dependent types, the universe hierarchy, and inductive types, all distinctive features of CIC, require a careful treatment.

Conceiving a gradual counterpart to CIC is however an interesting problem for multiple reasons. From the practical point of view, it could make programming in Coq and other proof assistants easier, by allowing the programmer to use the flexibility added by the gradual approach. From the theoretical point of view, it is an important test for gradual typing to try and apply it to a type system as complex as CIC. On the other side, gradualization requires some tools that do not belong to vanilla CIC, such as errors, making Gradual CIC a very interesting use case for the recent work on those.

Our contribution is to give a first description of a Gradual Calculus of Inductive Constructions, decomposed in two phases. In a first phase, we extend CIC to a gradual type system GCIC, by modifying its typing rules. Terms in this new system are compiled back to terms in CIC by using a few axioms. The second phase aims at giving a semantics, and in particular a computational content, to those axioms. Our main tool there are the recently developed notion of syntactical models.

There are still some technical details pertaining to general inductive types and the universe hierarchy to make precise, but once those are cleared, this contribution should make it into a full-fledged paper.

OUTLINE OF THE REPORT Section 1 presents CIC, and Section 2 presents gradual typing. Both aim at giving quick presentations of those two lines of work, and thus contain mostly classical results. They are as self-contained as possible, and hopefully complete enough to enable the reader to follow the rest of the report. Section 3 presents the first phase towards the gradualization of CIC, as we just described above. Section 4 presents the second phase. Those two sections constitute the core of this report, and, apart from Section 4.1, are mostly novel work. We tried to keep a balance between avoiding too much technicalities, while still exposing in some details the crucial points. Finally, Section 5 gives some concrete examples of the system described in the two previous sections. Although it lies at the very end, the reader might wish to look at it earlier in order to get a feeling of what we wish to achieve. Of particular interest is the discovery we expose there that the universe hierarchy prevents non-terminating behaviour usual of gradual type systems.

# 1. The Calculus of Inductive Constructions

The Calculus of Inductive Constructions is a type system that consists of two main ingredients. The first one is the Calculus of Constructions ($CC_\omega$), a formalism introduced by Coquand and Huet [CH88] which can be seen at the same time as a higher order logic and as a programming language with polymorphic types, corresponding to each other via the Curry-Howard isomorphism. $CC_\omega$ fits in the general picture of Pure Type Systems [Bar91], and can thus be seen as a (very strong) extension of the simply typed lambda calculus. It is already very powerful and expressive enough to encode inductively defined datatypes, for instance integers. However, this encoding is not quite satisfactory, hence the introduction of the second ingredient: inductive types as a primitive construction, as introduced by Paulin-Mohring [Pau93].

In Section 1.1, we give a quick introduction/recap of $CC_\omega$, and present inductive types in Section 1.2, following [Pau15]. Apart from making the report self-contained, we also wish to present the precise version of CIC we use, as there are a huge number of variants for it.

## 1.1. The Calculus of Constructions

As we already hinted, the Calculus of Constructions is a version on steroids of the simply typed lambda calculus (STLC). In STLC, the only binding allowed is terms binding other terms, and types exist mainly to prevent reduction from diverging. In $CC_\omega$, in contrast, terms and types have bindings, both for terms and types. This makes types and terms much more similar, and they are given by the same grammar. To be able to have the property that any term has a type, and avoid paradoxes linked with circularity, there is also a need for a hierarchy of universes $\square_i$ indexed by integers, each being the type of the previous. The notion of type becomes a consequence of typing: a type is anything which has type $\square_i$ for some $i$.

In Figure 1 we give the grammar to form terms $t$, contexts $\Gamma$, as well as the typing rules: $\vdash \Gamma$ means that the context $\Gamma$ is well-formed, and $\Gamma \vdash t : T$ means that in the context $\Gamma$ the term $t$ has type $T$. Finally, the relation $t \equiv t'$ states that the terms $t$ and $t'$ are convertible, i.e. they should be seen as equal via calculation, just as

for $\beta$-equivalence in STLC. Similarly to STLC, the main rule is $\beta$-reduction – although more will be added when extending $\mathrm{CC}_\omega$ to CIC and beyond. It can be performed anywhere in a term, hence the congruence closure.

**Terms:**

$$\begin{array}{rcll} t ::= & x & & \text{variable} \\ | & \Box_i & & \text{universe} \\ | & t\ t & & \text{application} \\ | & \lambda\, x : t.t & & \text{abstraction} \\ | & \Pi\, x : t.t & & \text{dependent function type/universal quantification} \end{array}$$

**Contexts:**

$$\Gamma := \cdot \mid x : t, \Gamma$$

**Typing:**

$$\frac{}{\vdash \cdot} \qquad \frac{\Gamma \vdash A : \Box_i}{\vdash \Gamma, x : A} \qquad \frac{\Gamma \vdash B : \Box_i \qquad \Gamma \vdash x : A}{\Gamma, y : B \vdash x : A} \qquad \frac{\vdash \Gamma, x : A}{\Gamma, x : A \vdash x : A} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \Box_i : \Box_{i+1}}$$

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash \Pi\, x : A.B : \Box_i}{\Gamma \vdash \lambda\, x : A.t : \Pi\, x : A.B} \qquad \frac{\Gamma, x : A \vdash B : \Box_i \qquad \Gamma \vdash A : \Box_j}{\Gamma \vdash \Pi\, x : A.B : \Box_{\max(i,j)}} \qquad \frac{\Gamma \vdash t : \Pi\, x : A.B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B\{x := u\}}$$

$$\frac{\Gamma \vdash t : A \qquad A \equiv B \qquad \Gamma \vdash B : \Box_i}{\Gamma \vdash t : B}$$

**Conversion:**

$$(\lambda\, x : A.t)\ u \mapsto_\beta t\{x := u\} \qquad \text{+ rules for congruence closure}$$

**Figure 1:** Grammar and typing rules of the Calculus of Constructions

We did not include all the features of the theory of Coq, notably the impredicative sort Prop, in order to focus on the core features of CIC we mentioned in the introduction, and simplify the presentation a bit. To improve readability further, we use a few notational shortcuts:

- if the index of $\Box_i$ is not crucial, we drop it, writing only $\Box$
- we replace $\Pi\, x : A.B$ by $A \to B$ if $x$ does not appear in $B$ (non-dependent function type)
- we compress multiple $\lambda$ and $\Pi$ binders as one, e.g. we write $\lambda(x : A), (y : B).t$ rather than $\lambda\, x : A.\, \lambda\, y : B.t$

The system $\mathrm{CC}_\omega$ has many good properties, among which we want to mention two, as the wish to keep them will guide some choices in the next sections.

**Proposition 1 (Decidable typing)**
Given a term $t$ and a context $\Gamma$, the two following problems are decidable:

- type inference: given a term $t$ and a context $\Gamma$, is there a term $T$ such that $\Gamma \vdash t : T$ holds?
- type checking: given terms $t$ and $T$ and context $\Gamma$, is it true that $\Gamma \vdash t : T$ holds?

A crucial intermediate property is that $\equiv$ is decidable, which in turn ensues from strong normalization and confluence of $\mapsto_\beta$. Thus, strong normalization is an important feature, as decidable type checking is a very desirable feature when the type system under design is to be implemented.

**Proposition 2 (Consistency)**
Not all types are inhabited. In particular, there is no term of type $\Pi\, x : \Box.x$.

This property corresponds to the logical soundness of the system, as the type $\Pi\, x : \Box.x$ is a way to represent falsehood. Indeed, following the Curry-Howard isomorphism – and viewing $\Pi$ as a universal quantification – it states that all propositions are true, similarly to the principle of explosion of logic.

## 1.2. Inductive Types

IMPREDICATIVE DEFINITIONS The system $\mathrm{CC}_\omega$ as just presented is already extremely powerful, both in terms of which functions are definable, and of which logical predicates are expressible. In particular, one can give a so-called impredicative encoding of usual data-structures, similar to what is done in untyped lambda calculus: a data-structure is expressed as a (weakly) initial algebra for a signature. For instance booleans correspond to the type $\Pi\, X : \Box.X \to X \to X$, product of $A$ and $B$ to $\Pi\, X : \Box.(A \to B \to X) \to X$, natural numbers to $\Pi\, X : \Box.X \to (X \to X) \to X$, and so on. This works fine to do recursive definitions. Following our example, if $N$ is the type just given for natural numbers, a term $n$ of type $N$ can be used as a recursor, as follows

$$\lambda(A : \Box), (a : A), (f : A \to A), (n : N).n\ A\ a\ f : \Pi\, A : \Box.A \to (A \to A) \to N \to A$$

However, this encoding has a drawback: without further assumptions on the system, the impredicative encoding only gives a weakly initial algebra and not an initial algebra. This means that the impredicative encoding is not well-suited to do induction, and indeed in $\mathrm{CC}_\omega$ one cannot prove that the above $N$ satisfies the following induction principle (recall that $\Pi$ corresponds to universal quantification):

$$\Pi\, P : N \to \Box.P(z) \to (\Pi\, n : N.P\ n \to P\ (S\ n)) \to \Pi\, n : N.P\ n$$

where $z := \lambda(X : \Box), (x : X), (f : X \to X).x$ and $S := \lambda(n : N), (X : \Box), (x : X), (f : X \to X).f\ (n\ X\ x\ f)$ are the respective impredicative encodings of zero and successor.

INDUCTIVE TYPES This problem was noticed early on in the development of $\mathrm{CC}_\omega$, and an alternative was proposed in [Pau93]: introducing a new concept to the theory that enables well-behaved inductive definitions. Those are the so-called inductive types. A generic inductive definition looks like follows (in a syntax close to Coq, see below for an example, and Appendix A for more):

$$\begin{array}{l}\texttt{Inductive}\ \ I\ \ (x_1 : A_1) \ldots (x_m : A_m)\ \ (y_1 : B_1) \ldots (y_n : B_n)\ \ \texttt{:}\ \ \Box_i\ \ \texttt{:=} \\ \quad \texttt{|}\ \ c_1\ \ \texttt{:}\ \ \Pi(z_1^1 : U_1^1), \ldots, (z_{n_1}^1 : U_{n_1}^1).I\ \ x_1 \ldots x_m\ \ t_1^1 \ldots t_n^1 \\ \quad \vdots \\ \quad \texttt{|}\ \ c_k\ \ \texttt{:}\ \ \Pi(z_1^k : U_1^k), \ldots, (z_{n_k}^k : U_{n_k}^k).I\ \ x_1 \ldots x_m\ \ t_1^k \ldots t_n^k\end{array}$$

Such a definition adds to the syntax of $\mathrm{CC}_\omega$ the new terms

$$I : \Pi(x_1 : A_1), \ldots (x_m : A_m), (y_1 : B_1), \ldots (y_n : B_n).\Box_i$$

and

$$c_j : \Pi(x_1 : A_1), \ldots, (x_m : A_m), (z_1^1 : U_1^1), \ldots, (z_{n_1}^1 : U_{n_1}^1).I\ x_1 \ldots x_m\ t_1^1 \ldots t_n^1$$

The term $I$ is to be understood as a new type family, defined inductively by the constructors $c_j$. For such a definition to be correct, there are technical conditions related to well-typedness of the terms $t_i^j$, as well as on the way $I$ itself can appear in the $U_i^j$ (the so-called positivity condition). We do not describe them here, the details can be found in [Pau93], but these restrictions are crucial to ensure that the newly defined inductive types behave well, in particular that they do not compromise strong normalization of the system.

Along with $I$ and $c_j$, a last term $\mathrm{rec}_I$ is generated, that should be understood as an induction principle for $I$, expressing that $I$ is initial. Intuitively, $\mathrm{rec}_I$ takes a type $P$ depending on $I$ and says that to inhabit $P$ for an arbitrary inhabitant of $I$ it suffices to inhabit $P$ for all constructors. Giving an exact account of the shape of that principle is again quite technical. We also refer to [Pau93] for the details, and rather give a simple instance below, and more complex ones in Appendix A.

This principle goes with a new reduction rule, dubbed $\iota$-reduction and written $\mapsto_\iota$, saying that the term defined by $\mathrm{rec}_I$ reduces to the case given for constructor $c_j$ in case when used on a term with head constructor $c_j$. Again, see below and in Appendix A for examples.

These inductive definitions turn $\mathrm{CC}_\omega$ into an open system that can be extended with new inductive types. To remain in a closed syntax with a finite number of constructors we work with CIC, a system obtained by adding to $\mathrm{CC}_\omega$ a finite but arbitrary amount of well-formed inductive types. All definitions we give handle generic inductive types so that they work whatever we consider for CIC.

EXAMPLES OF INDUCTIVE TYPES A very simple concrete example of the previous generic definitions are the booleans, given by:

$$\begin{array}{l}\texttt{Inductive}\ \ \mathbf{B}\ \ \texttt{:}\ \ \Box\ \ \texttt{:=} \\ \texttt{|}\ \ \mathrm{true}\ \ \texttt{:}\ \ \mathbf{B} \\ \texttt{|}\ \ \mathrm{false}\ \ \texttt{:}\ \ \mathbf{B}\end{array}$$

The corresponding induction principle, with its computation rules, are

$$\mathrm{rec}_{\mathbf{B}} : \Pi\, P : \mathbf{B} \to \square.(P\ \mathrm{true}) \to (P\ \mathrm{false}) \to \Pi\, b : \mathbf{B}.P\ b$$

$$\mathrm{rec}_{\mathbf{B}}\ P\ t_{\mathrm{true}}\ t_{\mathrm{false}}\ \mathrm{true} \mapsto_\iota t_{\mathrm{true}} \qquad \mathrm{rec}_{\mathbf{B}}\ P\ t_{\mathrm{true}}\ t_{\mathrm{false}}\ \mathrm{false} \mapsto_\iota t_{\mathrm{false}}$$

In the next sections, we use multiple usual inductive types, both as examples of our framework and as tools: booleans $\mathbf{B}$, natural numbers $\mathbf{N}$, dependant sum $\Sigma\, x : A.B$, equality $\mathrm{Id}_A\ a\ a'$, vectors $\mathrm{Vect}\ A\ n$, unit/true type $\top$, empty/false type $\bot$. Their definitions using the previous syntax are presented in full in Appendix A, together with the obtained terms and reduction rules. When needed, we assume that the version of CIC we consider contains at least those.

ENCODING VIA EQUALITY An important distinction considering the arguments of inductive types is between parameters and indices. Let $I : \Pi(x_1 : A_1), \dots (x_m : A_m), (y_1 : B_1), \dots (y_n : B_n).\square_i$ be an inductive family, defined as above. The $x_i$ are called parameters and are uniform across constructors. The $y_i$, on the other hand, are called indices, and they depend on the constructor used. In particular, using the indices, one can possibly exclude impossible constructors. For instance, a term of type $\mathrm{Vect}\ A\ (\mathrm{S}\ n)$ cannot have been constructed with constructor $\mathrm{nil}$, as it constructs terms of type $\mathrm{Vect}\ A\ 0$.

There is however a generic way to encode any inductive type in an indices-free way, using equalities. The idea is to replace the indices with parameters, and to make all constructors take equalities between those parameters and the indices of the original inductive. Given an inductive

$$\begin{array}{l}\texttt{Inductive}\ \ I\ \ (x_1 : A_1) \dots (x_m : A_m)\ \ (y_1 : B_1) \dots (y_n : B_n)\ \ \texttt{:}\ \ \square_i\ \ \texttt{:=} \\ |\ \ c_1\ \ \texttt{:}\ \ \Pi(z_1^1 : U_1^1), \dots, (z_{n_1}^1 : U_{n_1}^1).I\ \ x_1 \dots x_m\ \ u_1^1 \dots\ \ u_n^1 \\ \vdots \\ |\ \ c_k\ \ \texttt{:}\ \ \Pi(z_1^k : U_1^k), \dots, (z_{n_k}^k : U_{n_k}^k).I\ \ x_1 \dots x_m\ \ u_1^k \dots\ \ u_n^k\end{array}$$

it can be transformed into

$$\begin{array}{l}\texttt{Inductive}\ \ I'\ \ (x_1 : A_1) \dots (x_m : A_m)\ \ (y_1 : B_1) \dots (y_n : B_n)\ \ \texttt{:}\ \ \square_i\ \ \texttt{:=} \\ |\ \ c'_1\ \ \texttt{:}\ \ \Pi(z_1^1 : U_1^1), \dots, (z_{n_1}^1 : U_{n_1}^1), (e_1^1 : \mathrm{Id}_{B_1^1}\ \ y_1\ \ \tilde{u}_1^1), \dots (e_n^1 : \mathrm{Id}_{B_n^1}\ \ y_n\ \ \tilde{u}_n^1).I\ \ x_1 \dots x_m\ \ y_1 \dots\ \ y_n \\ \vdots \\ |\ \ c'_k\ \ \texttt{:}\ \ \Pi(z_1^k : U_1^k), \dots, (z_{n_k}^k : U_{n_k}^k), (e_1^k : \mathrm{Id}_{B_1^k}\ \ y_1\ \ \tilde{u}_1^k), \dots (e_n^k : \mathrm{Id}_{B_n^k}\ \ y_n\ \ \tilde{u}_n^k).I\ \ x_1 \dots x_m\ \ y_1 \dots\ \ y_n\end{array}$$

where the $\tilde{u}_i^j$ are versions of the $u_i^j$ modified to be of type $B_i^j$, by using the equalities $e_i^l$. A concrete example is given for vectors in Appendix B.

To see why this transformation is sensible note that while the $y_i$ in the return type are now parameters, they are still linked through the equalities $e_i^j$ to their actual value $u_i^j$. In fact the new $I'$ and $c'_j$ primitives can be used to simulate $I$ and $c_j$. Indeed, terms $c''_j$ with the same type as the $c_j$ (with $I$ replaced with $I'$) can be constructed from the $c'_j$. Similarly, a term $\mathrm{rec}'_I$ of the same type as $\mathrm{rec}_I$ (with $I$ replaced with $I'$ and $c_j$ with $c''_j$) can be constructed from $\mathrm{rec}_{I'}$ by using the $e_i^j$. Moreover, this $\mathrm{rec}'_I$ has exactly the same computational behaviour as $\mathrm{rec}_I$.

We describe this transformation as we use it in Section 4.3, when we translate inductive types: instead of giving the translation on every inductive types, we only translate inductive types with parameters and the equality type, resorting to this transformation to encode any inductive type using those.

# 2. Gradual Typing

As we already explained, one of the most widely asked question in the gradual typing literature is "How do I turn this static system gradual?". Although there are of course a fair amount of variations between the answers, the mechanisms still usually have a somewhat common structure.

The first step is to extend the type system with a new base type $?$, that should be understood as a type that will only be checked at runtime, i.e. a dynamic type – similar to the `dynamic` keyword of C#. The typing judgement is then modified to account for this introduction, by relaxing the typing rules with an optimistic treatment of $?$. For instance, equality of types is replaced by a consistency relation $\sim$, saying that two types *could* be equal. This optimism introduced at typing time is then counterbalanced by the introduction of some checks that can fail at run-time, should the optimistic assumption be violated – as in dynamic typing.

In this section, we illustrate this process, using simply typed $\lambda$-calculus as a toy static type system, partly following [Sie+15]. In Section 2.1, we present the gradualization, then discuss the properties a gradual type system should respect in Section 2.2. Finally in Section 2.3, we give a quick look at other gradual type systems related to intermediate systems between simply typed lambda calculus and CIC.

## 2.1. Gradually Typed Lambda Calculus

SIMPLY TYPED LAMBDA CALCULUS The exact system STLC we want to gradualize is given in Figure 2, it is simply typed lambda calculus with two base types and a few constants (booleans, natural numbers, sum and if), in order to see how they are handled. We call the types $T$ defined here static, as opposed to the gradual types we construct afterwards.

**Types:** $T := \mathbf{B} \mid \mathbf{N} \mid T \to T$
**Terms:** $t := x \mid \underline{n} \mid \underline{b} \mid \lambda\, x : T.t \mid t\ t \mid t + t \mid \text{if } t \text{ then } t \text{ else } t$ with $n \in \mathbb{N}$ and $b \in \mathbb{B}$
**Typing:**

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{}{\Gamma \vdash \underline{n} : \mathbf{N}} \qquad \frac{}{\Gamma \vdash \underline{b} : \mathbf{B}} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda\, x : T_1.t : T_1 \to T_2}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2 \qquad \mathrm{dom}(T_1) = T_2}{\Gamma \vdash t_1\ t_2 : \mathrm{cod}(T_2)} \qquad \frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2 \qquad T_1 = \mathbf{N} \qquad T_2 = \mathbf{N}}{\Gamma \vdash t_1 + t_2 : \mathbf{N}}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2 \qquad \Gamma \vdash t_3 : T_3 \qquad T_1 = \mathbf{B} \qquad T_2 = T \qquad T_3 = T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$$

**Reduction:**

$$(\lambda\, x : T.t)\ u \mapsto_\beta t\{x := u\} \qquad \underline{n} + \underline{n'} \mapsto_+ \underline{n + n'} \qquad \text{if } \underline{\text{true}} \text{ then } t_1 \text{ else } t_2 \mapsto_{\text{if}} t_1 \qquad \text{if } \underline{\text{false}} \text{ then } t_1 \text{ else } t_2 \mapsto_{\text{if}} t_2$$

**Figure 2:** Simply Typed Lambda Calculus

The system itself is not very surprising, the only quite non-standard point is that instead of having the same type appear in multiple places, we explicit those relations using equalities. Because of that, we also use the $\mathrm{dom}$ and $\mathrm{cod}$ functions, that are defined such that $\mathrm{dom}(T_1 \to T_2) := T_1$, $\mathrm{cod}(T_1 \to T_2) = T_2$ and are undefined otherwise. In case they are not defined, the rule is not applicable.

The reduction $\mapsto$ is the contextual closure of the base reduction rules $\mapsto_\beta$, $\mapsto_+$ and $\mapsto_{\text{if}}$, i.e. reduction can happen at any place in a term.

GRADUALIZING TYPING The first step towards gradualization is to extend types to $T := \cdots \mid ?$, with $?$ representing a type we do not want to check at typing time. Term definition does not change in itself, but $\lambda$ abstractions now feature gradual types instead of static types. At typing time, we want to treat this $?$ in an optimistic way, i.e. we consider it could stand for any other type. Thus, we define the consistency relation, as follows:

**Definition 3 (Consistency)**
The consistency relation $\sim$ is inductively defined by the following rules:

$$\frac{}{T \sim T} \qquad \frac{}{? \sim T} \qquad \frac{}{T \sim ?} \qquad \frac{T_1 \sim T_2 \qquad T_1' \sim T_2'}{T_1 \to T_1' \sim T_2 \to T_2'}$$

Intuitively, two types are consistent if they *could* be equal, were the occurrences of $?$ to take the right values. Note that this relation is reflexive and symmetric but *not* transitive: if it were transitive then it would relate all types, as any type is consistent with $?$.

All the typing rules are then updated accordingly, replacing equality of types with the looser relation of consistency. The domain and codomain functions are updated to $\mathrm{dom}_?$ and $\mathrm{cod}_?$ along the same idea: considering that $?$ could be any type, its domain and codomain are both defined to be $?$. In the end, we get the typing rules of Figure 3, where the difference with the static rules have been highlighted.

With these rules, we get a new type system, dubbed Gradually Typed Lambda Calculus (GTLC), where typing is more flexible than in STLC. For instance, a term like $(\lambda\, x : ?.x + 1)\ \mathrm{true}$ typechecks (with type $\mathbf{N}$), because $?$ is consistent with both $\mathbf{N}$ and $\mathbf{B}$. Of course, even if this term is type correct, we cannot just $\beta$-reduce it, because this would lead to a term $\mathrm{true} + 1$ that is not well-typed any more. Thus, we need to change the semantics as well, if we want to ensure that typing is preserved by reduction – a property usually known as subject reduction.

$$\frac{(x:T)\in\Gamma}{\Gamma\vdash x:T} \qquad \frac{}{\Gamma\vdash \underline{n}:\mathbf{N}} \qquad \frac{}{\Gamma\vdash \underline{b}:\mathbf{B}} \qquad \frac{\Gamma,x:T_1\vdash t:T_2}{\Gamma\vdash \lambda\, x:T_1.t:T_1\to T_2}$$

$$\frac{\Gamma\vdash t_1:T_1 \qquad \Gamma\vdash t_2:T_2 \qquad \mathbf{dom_?}(T_1)\sim T_2}{\Gamma\vdash t_1\ t_2:\mathbf{cod_?}(T_1)} \qquad \frac{\Gamma\vdash t_1:T_1 \qquad \Gamma\vdash t_2:T_2 \qquad T_1\sim\mathbf{N} \qquad T_2\sim\mathbf{N}}{\Gamma\vdash t_1+t_2:\mathbf{N}}$$

$$\frac{\Gamma\vdash t_1:T_1 \qquad \Gamma\vdash t_2:T_2 \qquad \Gamma\vdash t_3:T_3 \qquad T_1\sim\mathbf{B} \qquad T_2\sim T \qquad T_3\sim T}{\Gamma\vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3:T}$$

**Figure 3:** Typing rules of Gradually Typed Lambda Calculus

COMPILATION TO THE CAST CALCULUS As we just highlighted, we cannot just reuse the reduction of STLC for GTLC if we want to ensure subject reduction – an essential property of any type system. The problem is that we cannot just forget in terms that consistency was used to type them. That use needs to be reflected in the terms, in order to keep track of the fact that we were optimistic during typing, enabling us to check at "runtime" that the optimistic assumptions made by consistency indeed hold.

We therefore compile well-typed terms of GTLC to an extension of STLC, called cast calculus, where we keep explicit track of the places where consistency was used, using a new $\mathrm{cast}$ operator. This compilation is directed by the typing derivation of the term of GTLC. The syntax of the cast calculus is given in Figure 4, as well as some rules of the compilation, denoted as $\rightsquigarrow$, to give a flavour of the way it works. The whole compilation and the typing rules are detailed in Appendix C. The most important thing about the typing rules is that they are static, insofar as they do not resort to consistency, and treat ? as just another base constant.

**Types:** $T := \mathbf{B} \mid \mathbf{N} \mid ? \mid T \to T$
**Terms:** $t := x \mid \underline{n} \mid \underline{b} \mid \lambda\, x:T.t \mid t\ t \mid t+t \mid \text{if } t \text{ then } t \text{ else } t \mid \mathrm{raise} \mid \mathrm{cast}_{T,T}\ t$ with $n\in\mathbb{N}$ and $b\in\mathbb{B}$
**Compilation from GTLC (extract):**

$$\frac{}{\Gamma\vdash \underline{n}\rightsquigarrow \underline{n}:\mathbf{N}} \qquad \frac{\Gamma\vdash t_1\rightsquigarrow t_1':T_1 \qquad \Gamma\vdash t_2\rightsquigarrow t_2':T_2 \qquad T_1\sim\mathbf{N} \qquad T_2\sim\mathbf{N}}{\Gamma\vdash t_1+t_2\rightsquigarrow (\mathrm{cast}_{T_1,\mathbf{N}}\ t_1')+(\mathrm{cast}_{T_2,\mathbf{N}}\ t_2'):\mathbf{N}}$$

$$\frac{\Gamma\vdash t_1\rightsquigarrow t_1':T_1 \qquad \Gamma\vdash t_2\rightsquigarrow t_2':T_2 \qquad \mathrm{dom}_?(T_1)\sim T_2}{\Gamma\vdash t_1\ t_2\rightsquigarrow (\mathrm{cast}_{T_1,\mathrm{dom}_?(T_1)\to\mathrm{cod}_?(T_1)}\ t_1')\ (\mathrm{cast}_{T_2,\mathrm{dom}_?(T_1)}\ t_2'):\mathrm{cod}_?(T_2)}$$

**Figure 4:** The Cast Calculus (Extract)

REDUCTION IN THE CAST CALCULUS Now that the coercions have been inserted, the rules $\mapsto_\beta$ and $\mapsto_{\text{if}}$ satisfy subject reduction. However, we also need reduction rules for our two new operators $\mathrm{cast}$ and $\mathrm{raise}$. For $\mathrm{raise}$, the rules are fairly easy: since we do not have any error catching mechanism, $\mathrm{raise}$ just propagates. For $\mathrm{cast}_{T_1,T_2}\ t$, the rules are given in Figure 5.

| | | |
|---|---|---|
| Success: | $\mathrm{cast}_{T_1,T_2}\ t\mapsto_{\mathrm{cast}} t$ | if $T_1$ and $T_2$ are $\mathbf{B}$, $\mathbf{N}$ or ? and $T_1=T_2$ |
| Failure: | $\mathrm{cast}_{T_1,T_2}\ t\mapsto_{\mathrm{cast}} t$ | if $T_1$ and $T_2$ are both $\mathbf{B}$, $\mathbf{N}$ or $\cdot\to\cdot$ and have different heads |
| Function: | $\mathrm{cast}_{T_1'\to T_1'',T_2'\to T_2''}\ t\mapsto_{\mathrm{cast}} \lambda\, x:T_2'.\ \mathrm{cast}_{T_1'',T_2''}\ t\ (\mathrm{cast}_{T_2',T_1'}\ x)$ | |
| Cast cancellation: | $\mathrm{cast}_{?,T_2}\ \mathrm{cast}_{T_1,?}\ t\mapsto_{\mathrm{cast}} \mathrm{cast}_{T_1,T_2}\ t$ | |

**Figure 5:** Reduction of the casting operator

The first two rules are the base cases: the cast disappears in case the types are equal base static types, and fails if they have different static head constructors. Otherwise, the third rule applies, recursively decomposing a cast between arrow types into a cast on the input and a cast on the output. The last rule is the most interesting: it enables to reveal the type of a term hidden under type ? as it is used. It is in a way similar to looking up the type tag of a value at runtime in a dynamic typing setting. The term $\mathrm{cast}_{T_1,?}\ t$ can indeed be seen as the term $t$ together with a type tag $T_1$, and casting this term to the type $T_2$ triggers a comparison between $T_1$ and $T_2$ in order to check that this cast is legal.

## 2.2. Properties of Gradual Type Systems

The main properties expected from a gradual type system are twofold. First, there should be an embedding of the corresponding fully static and fully dynamic systems into the gradual type system, in order to ensure that the gradual system really is a superset of those. But these properties do not say anything of what happens in between those two extremes. This is the reason why [Sie+15] introduced the so-called gradual guarantee, which says what should happen when relaxing parts of a term from the static to the looser dynamic discipline. We give those properties in the case of GTLC as an illustration.

EMBEDDINGS We denote as $\vdash_S$ the typing judgement of STLC, and write $t \Downarrow_S t'$ if $t'$ is a normal form for $t$ in STLC. Then we have the following:

**Proposition 4 (Correctness of the embedding of STLC)**
If $t$ is a term of STLC (that can also be seen as a term of GTLC without any ?) and $T$ is a type of STLC then

- $\vdash_S t : T$ iff $\vdash t : T$
- for any term $v$, $e \Downarrow_S v$ iff $e \Downarrow v$

Which says that STLC really is a subsystem of GTLC, both from the typing and the reduction point of view.

The embedding for the other end of the spectrum is a little bit more involved: we want to consider pure lambda calculus (PLC), given by the following syntax:

$$t := x \mid \underline{n} \mid \underline{b} \mid \lambda\, x.t \mid t\ t \mid t + t \mid \text{if } t \text{ then } t \text{ else } t$$

The difference with STLC is that abstraction does not bear a type. We keep the same reduction rules as STLC, and of course that all terms are valid, as there are no types. We embed those terms into GTLC, as follows:

**Definition 5 (Embedding of pure lambda calculus in GTLC)**
The embedding $\lceil \cdot \rceil$ is defined recursively on term of PLC as follows:

$$\lceil x \rceil := x \qquad \lceil n \rceil := n :: ? \qquad \lceil b \rceil := b :: ? \qquad \lceil \lambda\, x.t \rceil := (\lambda\, x : ?\,.\lceil t \rceil) :: ? \qquad \lceil t\ t' \rceil := \lceil t \rceil\ \lceil t' \rceil$$

$$\lceil t + t' \rceil := (\lceil t \rceil + \lceil t' \rceil) :: ? \qquad \lceil \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rceil := \text{if} \lceil t_1 \rceil \text{ then} \lceil t_2 \rceil \text{ else} \lceil t_3 \rceil$$

where $t :: T$ is a shortcut for $(\lambda\, x : T.x)\ t$, so that $t :: T$ always has type $T$.

We write $t \Downarrow_P v$ if $v$ is a normal form for $t$ in PLC, and we have the following:

**Proposition 6 (Correctness of the embedding of PLC)**
If $t$ is any term of PLC, we have the following:

- if $t$ has no free variable, then $\vdash \lceil t \rceil : ?$
- if $t \Downarrow_P v$, then either $\lceil t \rceil \Downarrow \text{cast}_{T,?}\, v$ for some $T$ or $\lceil t \rceil \Downarrow \text{raise}$
- if $\lceil t \rceil \Downarrow v$ then $t \Downarrow_P v'$ and $v = \text{cast}_{T,?}\, v'$ for some $T$

This is similar to the embedding of STLC, apart from the fact that terms of PLC can be stuck because of a type error, that causes their GTLC counterpart to raise an error.

GRADUAL GUARANTEE These properties give constraints on both ends of the spectrum. The key property to link those ends is the so-called gradual guarantee. It is the core of the gradual approach, characterizing the relation between the dynamic and static types. To state it, we need the following:

**Definition 7 (Type precision)**
The type precision ordering between types, written $\sqsubseteq$, is given by the following rules:

$$\frac{}{? \sqsubseteq T} \qquad \frac{}{\mathbf{N} \sqsubseteq \mathbf{N}} \qquad \frac{}{\mathbf{B} \sqsubseteq \mathbf{B}} \qquad \frac{T_1 \sqsubseteq T_2 \qquad T_1' \sqsubseteq T_2'}{T_1 \to T_1' \sqsubseteq T_2 \to T_2'}$$

The relation is extended to terms, by saying a term $t$ of GTLC is more precise than another $t'$ if they have the same shape and if all types in the abstractions of $t$ are more precise than those in $t'$.

Intuitively, it says that $t$ and $t'$ are the same program, but with $t$ having more static type annotations than $t'$. Now the gradual guarantee goes as follows:

**Proposition 8 (Gradual Guarantee)**
Suppose $t$ and $t'$ are terms of GTLC such that $t \sqsubseteq t'$ and $\vdash t : T$. Then:

1. there is a type $T'$ such that $\vdash t' : T'$ and $T \sqsubseteq T'$
2. if $t \Downarrow v$, then $t' \Downarrow v'$ with $v \sqsubseteq v'$, and if $t \Uparrow$ then $t' \Uparrow$
3. if $t' \Downarrow v'$ then either $t \Downarrow v$ with $v \sqsubseteq v'$ or $t \Downarrow \mathrm{raise}$, and if $t' \Uparrow$ then either $t \Uparrow$ or $t \Downarrow \mathrm{raise}$

The first property says that relaxing types cannot cause terms to fail typechecking. This is crucial, as the purpose of the gradual approach is to ensure a smooth transition between the dynamic and static worlds. The gradual guarantee ensures this: if the term on the static end typechecks, then every step of the transition also typechecks. There is no need for coordinated type annotations in different place, or to make efforts in order to understand how the typechecker works. The second and third properties give similar guarantees on the execution.

## 2.3. Beyond Simply Typed Lambda Calculus

Quite a few papers already handle some of the characteristic of CIC in a gradual setting. We give a short survey of those here.

In [TT15], a first approach to dependent types is given, but only refinement types (i.e. types $\Sigma\, x : A.P\ x$ for some proposition $P$) are studied, and the focus is set on decidable properties. The interest for decision is also present in [DTT18], where the focus is set on the relation between indexed and unindexed datatypes. In the same line of work, [LT17] also focuses on refinement types. We found that we share some of their concerns with decidability when trying to handle identity types (see Section 4.3), however refinement types are a too specific kind of dependent types to give real insight for general $\mathrm{CC}_\omega$, let alone CIC.

In [TLT19], a gradualization of System F – an intermediate between STLC and $\mathrm{CC}_\omega$ – is given, however the main challenge there is to preserve parametricity, a property that $\mathrm{CC}_\omega$ does not enjoy without further addition. Thus, it is not really relevant in our setting.

Finally, [ETG19] is maybe the most interesting approach to the question, as it considers $\mathrm{CC}_\omega$ in full. Although inductive types are not considered at all there, their approach to consistency can be of use in our context – see Section 3.1 for details.

# 3. Type System for Gradual CIC

Similarly to STLC, we separate the gradualization of CIC into two phases. The first one corresponds to the gradual typing discipline and to the insertion of a cast operator during typing to recover a statically well-typed

term, while the second is studies the semantics of that cast operator. This section is devoted to the first part, and the next tackles the second.

## 3.1. Gradually Typed CIC

To extend the typing rules of CIC into a gradually typed system GCIC, we follow the same roadmap as for STLC:

1. extend the syntax with a ? constructor
2. modify the typing rules to use consistency rather than convertibility of types
3. define a meaningful consistency relation

EXTENDING THE SYNTAX Because CIC does not make a syntactic difference between types and terms, we simply extend the term definition:

$$t := \cdots \mid ?$$

UPDATING THE TYPING A nice feature of CIC is that its typing rules already incorporates a notion of comparison between types, up to conversion. So we just have to add the more permissive

$$\frac{\Gamma \vdash t : A \qquad A \sim B \qquad \Gamma \vdash B : \square}{\Gamma \vdash t : B}$$

that makes use of the consistency relation $\sim$ instead of the conversion. Since $\sim$ is weaker than $\equiv$, we could keep only this rule without changing the typable terms in our new system. However, every use of this rule will add a $\mathrm{cast}$ operation when compiled (similarly to GTLC), and we wish to avoid it when it is not needed. Thus we keep the usual conversion rule of CIC, that treats $?$ as a constant without any special computational properties. Because $?$ is a term, we also need a rule to type it, which is simply

$$\frac{\Gamma \vdash T : \square}{\Gamma \vdash ? : T}$$

meaning that $?$ as a term can inhabit any type. Indeed, we want to use $?$ as any unknown part of a type, which can be any term, since we work with dependent types. Typically, we want to consider an inductive type with unknown index, such as $\mathrm{Vect}\ A\ ?$. Thus $?$ must be usable as a placeholder at any type, not just at $\square$.

AXIOMATIC CONSISTENCY Those modifications are quite straightforward, but the real complexity is hidden in the consistency relation. Its definition is not as simple as for GTLC, for at least two reasons. The first one is that consistency has to be defined on all terms, not only types, as $?$ can appear in any position. The second is that it cannot be simply defined inductively on syntax as in GTLC, as we want consistency to take that computation into account, and thus be at least as strong as conversion.

Another strong requirement is that we want consistency to be decidable, to ensure that typing stays so. This is in tension with the previous points, since a theoretically satisfying definition of consistency might be "too semantical" to be decidable. An example is given in the next paragraph.

Looking at this in another way, however, there is also some freedom in definition on the choice of consistency. Thus, we express in an axiomatic way which properties a consistency should have, rather than define a precise one. We first need a notion of precision, with respect to which a consistency is defined.

**Definition 9 (Acceptable precision)**
An acceptable precision $\sqsubseteq$ is a binary relation between terms of GCIC such that:

1. $\sqsubseteq$ is a preorder
2. $\sqsubseteq$ contains conversion, i.e. if $t \equiv t'$ then $t \sqsubseteq t'$
3. if $t$ and $t'$ are static terms, then $t \sqsubseteq t'$ iff $t \equiv t'$

**Definition 10 (Acceptable consistency)**
An acceptable consistency $\sim$ with respect to an acceptable precision $\sqsubseteq$ is a binary relation between terms of GCIC such that:

1. $\sim$ is reflexive and symmetric
2. $\sim$ is monotone with respect to $\sqsubseteq$, i.e. if $t \sim t'$ and $t' \sqsubseteq t''$ then $t \sim t''$
3. if $t$ and $t'$ are static terms, then $t \sim t'$ iff $t \equiv t'$

Note that that the monotony and reflexivity entail that an acceptable consistency must contain conversion. But monotony is much stronger, and it is the key ingredient ensuring the "typing" part of the gradual guarantee.

CONCRETE CONSISTENCIES The smallest precision is simply conversion, and the smallest acceptable consistency (with respect to that precision) is also conversion. This consistency yields exactly the same typable terms as CIC. On the other end of the spectrum, the biggest precision is the precision that is conversion on static terms and such that any non-static term is greater than any other term. The greatest acceptable consistency (with respect to that precision) says that two terms are consistent if at least one of them is non-static, or if they are convertible. This consistency defers all type comparisons between non-static types to the cast operator. Of course, those two extremes cannot really be called gradual: the first lacks any kind of dynamic features, while the second does almost no static type checks. But they give an idea of what a consistency relation does: for each pair of types it arbitrates between rejecting it at typing time or accepting it and risking cast errors.

A more interesting set of definitions is based on substitutions:

**Definition 11 (Substitution precision, substitution consistency)**
A substitution $\sigma$ is a mapping from each $?$ to a term.
The substitution precision $\sqsubseteq_s$ is defined as follows: $t \sqsubseteq_s t'$ if for every substitution $\sigma$ there is a substitution $\sigma'$ such that $\sigma(t) \equiv \sigma'(t')$.
The substitution consistency $\sim_s$ is defined as follows: $t$ and $t'$ are consistent if they are unifiable, i.e. if there is a substitution $\sigma$ such that $\sigma(t) \equiv \sigma(t')$.
The substitution precision is acceptable, and the substitution consistency is acceptable with respect to it.

These somewhat captures the informal semantics of $?$, and they are quite close to the Abstract Gradual Typing approach [GCT16]. However the problem of higher order unification is undecidable [Dow01], so that basing a definition of GCIC on substitution consistency makes typing undecidable.

Because the substitution consistency seems a good theoretical definition of consistency, a good aim for a consistency relation is to give a decidable approximation of it. This approximation can lean on the conservative side, typically by resorting to a unification algorithm in the flavour of [ZS17] to try and find a suitable substitution. This solution, however, is not monotone with respect to the substitution precision, as it may fail to find a substitution that exists. On the other hand, the approximation can be too permissive, by allowing types to be consistent even when they are not unifiable. This a way to interpret the approximate normalization of [ETG19]. In this paper, they in particular prove that their consistency relation is acceptable with respect to substitution precision.

Again, we do not wish to make a definitive choice, and leave the arbitration open. In particular, the different options could be compared in case of an implementation.

## 3.2. Cast Insertion

As for STLC, once a term has a typing derivation, this derivation can be used to transform the term back to a statically typed term in a calculus with new primitives for casting. Thus we extend CIC with two new constants, yielding the system $\text{CIC}_{\text{cast}}$ described in Figure 6.

For now those constants are abstract axioms, and the next section is devoted to giving them a precise semantics. However, we can already give an intuition for it. The axiom $\mathrm{cast}$ is very similar to the one of the Cast Calculus for STLC: it should behave like the identity if its two first arguments are the same, and fail otherwise. The axiom $?$ is a bit more involved: used as a term (i.e. on the left of a colon) it should be thought

$$t := \cdots \mid ? \mid \mathrm{cast} \qquad \frac{}{\vdash ? : \Pi\, A : \square.\square} \qquad \frac{}{\vdash \mathrm{cast} : \Pi(A:\square),(B:\square),A \to B}$$

**Figure 6:** Syntax and typing extension of $\mathrm{CIC}_{\mathrm{cast}}$

$$\frac{\Gamma \vdash T : \square \rightsquigarrow T'}{\Gamma \vdash ? : T \rightsquigarrow ?\ T'} \qquad \frac{\Gamma \vdash t : A \rightsquigarrow t' \qquad A \sim B \qquad \Gamma \vdash A : \square \rightsquigarrow A' \qquad \Gamma \vdash B : \square \rightsquigarrow B'}{\Gamma \vdash \mathrm{cast}_{A',B'}\, t'}$$

**Figure 7:** Compilation rules for GCIC (Extract)

of as an error — the cast operator will reduce to it when its first arguments are not equal. However, when used as a type (i.e. on the right of a colon) $?$ $\square$ has the same semantics as $?$ in GTLC: it represents a type that should be checked at runtime.

Once we have the target, we can compile any typing derivation of GCIC into a derivation of $\mathrm{CIC}_{\mathrm{cast}}$. We give in Appendix D the whole set of compilation rules. The two most important are given in Figure 7, corresponding to the new typing rules associated respectively with $?$ and $\sim$. The other rules are mostly compositional.

## 3.3. Instrumenting Refinement Algorithms

To actually insert casts in a real life setting, resorting to the mechanism of the previous section is quite unfeasible, as it requires a fully annotated term to work on. Instead, a possibility for a concrete implementation is to instrument a refinement algorithm.

Those algorithms are in charge, in a proof assistant, to transform a term given by the user to a term that can be fed to the core of the proof assistant. Initially, they were mostly devoted to inferring the types that were not given by the user, but in modern proof assistants they have a much broader role, see [Asp+12] for some examples. In particular, a recurring feature of modern refinement algorithm is to silently insert user-defined coercions: for instance, the user can declare a coercion from the type of groups to type $\square$, mapping a group to its carrier, or from the type $\mathbf{N}$ to the type of reals, that will be inserted silently by the refinement algorithm to bridge the gap each time a group (resp. natural number) is used when a type (resp. real) was expected.

This usual use of coercions is very different from our cast. However, the mechanism to insert them, as described in [Asp+12], is quite close to the one we want: whenever a term $t$ has a certain type $A$ and needs to be used at another type $B$, the system checks if there is an existing coercion between those two types, and in that case adds that coercion around $t$ to turn it into a term of type $B$. If instrumented correctly, this functionality could be used to implement a much better compilation from GCIC to $\mathrm{CIC}_{\mathrm{cast}}$: it would avoid having to implement the feature by hand, which is quite a heavy work, and would also enable the user to work with GCIC together with a functional refinement algorithm, making it much more practical.

## 3.4. Gradual Theorems

From those definitions, we cannot give any theorem on the reduction of the terms, as for now $\mathrm{cast}$ is just a stuck axiom. However, we can already get the embedding of CIC and the typing part of the gradual guarantee. The central property is the following, ensuring that the cast insertion is well-behaved:

**Proposition 12 (Correctness of cast insertion)**
If $x_1 : A_1, \ldots, x_n : A_n \vdash t : T$ in GCIC, then $x_1 : A'_1, \ldots, x_n : A'_n \vdash t' : T'$ where $x_1 : A_1, \ldots, x_n : A_n \vdash t : T \rightsquigarrow t'$ and $A'_i$ and $T'$ are obtained by cast insertion in $A_i$ and $T$, respectively.

**Proof (Sketch)**
Intuitively, the proof is by induction on the derivation of $\Gamma \vdash t : T$: we can transform a derivation tree for $t$ in GCIC into a derivation tree with almost the same shape in $\mathrm{CIC}_{\mathrm{cast}}$, the only difference being the place where consistency was used in GCIC and a cast was used instead in $\mathrm{CIC}_{\mathrm{cast}}$. However, to completely carry the proof, there are some technical subtleties to handle.
In particular, we need a lemma saying that we can choose a set of canonical typing derivations for every derivable judgement $\Gamma \vdash t : T$, such that a sub-derivation of a canonical derivation is itself a derivation. The aim is to ensure that every

time we need to type $x_1 : A_1, \ldots, x_{i-1} : A_{i-1} \vdash A_i : \square$ the $A'_i$ we get is the same. This lemma is a consequence of decidability of typing: as long as we have a deterministic procedure for type checking, the derivation trees given by it are a system of canonical derivation. The induction is then performed on canonical derivations rather than on arbitrary derivations.

For the static part, we get:

**Proposition 13 (Embedding of CIC in $\text{CIC}_{\mathbf{cast}}$)**
If $t$ and $T$ are terms of CIC, then $\vdash t : T$ is derivable in CIC iff it is derivable in GCIC.

**Proof (Sketch)**
The proposition is a consequence of the correctness property of cast insertion, and of the fact that an acceptable consistency corresponds to conversion on static terms.

For the fully dynamic, we do not have an equivalent to the theorems on the embedding of pure lambda calculus, as we lack a system that could play the role of pure lambda calculus, i.e. a form of "untyped CIC".

For the gradual part, however, we get what we want:

**Proposition 14 (Gradual guarantee)**
Let $t$, $s$ and $T$ be three terms of GCIC, such that $\vdash t : T$ in GCIC and $t \sqsubseteq s$. Then we have that also $\vdash s : S$ with $T \sqsubseteq S$.

**Proof (Sketch)**
The key property here is that $\sim$ is monotone wrt. $\sqsubseteq$, so that any use of the consistency rule to type $t$ is still correct in order to type $s$.

# 4. Realizing the Cast Operator

In this section, we concentrate on the way to give a semantics to $\text{CIC}_{\text{cast}}$ by giving one to the axioms ? and $\mathrm{cast}$ of the previous section. To do so, we resort to syntactical models, and in particular to the so-called exceptional type theory. The main feature of those syntactical models is to give us a way to handle exceptions inside CIC. This is a challenging problem, because effects mess up heavily with dependent types by exposing calling conventions, and because naïve exceptions break consistency (because any type can be inhabited using exceptions).

Section 4.1 describes the general technique of syntactical models, Section 4.2 gives an overview of the way we use them, Section 4.3 gives a syntactical translation of the cast operator in an intermediate type system, which is in turn justified via another syntactical model in Section 4.4.

## 4.1. Syntactical Models

Syntactical models of type theory, first described in [BPT17], are a way to give a semantics to a type theory using type theory itself. In general, to give a semantics and/or justify a type theory, one gives a model, i.e. objects in some meta-theory that interpret the syntax under consideration. Syntactical models are a very special kind of models, insofar as most objects in the source theory $\mathcal{S}$ are interpreted as the same objects in a target, usually simpler, type theory $\mathcal{T}$: a context is interpreted by a context, a term by a term, conversion by conversion, and typing should be preserved as well. Another way to look at syntactical models is to see them as a kind of compilation, transforming a program in $\mathcal{S}$ into a program in $\mathcal{T}$.

An important use of those syntactical models is to justify $\mathcal{T}$ extended with some axiom. To do so, first find a suitable translation from $\mathcal{T}$ to itself, then find a term $t$ that inhabits the translation of the axiom's type. The consistency of the original $\mathcal{T}$ entails the consistency of $\mathcal{T}$ augmented by the axiom. But it is even better: because $t$ is a term, it has a computational behaviour, and is not just a stuck axiom. This computational behaviour can then be taken as a *definition* for the computational rules of the axiom in the source theory.

Used this way, syntactical translations are a dependently typed generalization of the idea underlying the CPS translation [Gri90], that realizes classical axioms in intuitionistic predicate logic by using a program transformation similar to the CPS transformation used in some compilers.

THE "$\times\mathbf{B}$" MODEL A simple example, to get an idea of what happens when defining such syntactical models, is a translation from $\mathrm{CC}_\omega$+Id to $\mathrm{CC}_\omega$+Id+$\mathbf{B}$+$\Sigma$, that justifies the negation of function extensionality in $\mathrm{CC}_\omega$. Function extensionality is the type

$$\mathrm{funext} := \Pi(A : \Box), (B : \Box), (f : A \to B), (g : A \to B).(\Pi\, x : A.f\ x = g\ x) \to f = g$$

saying that functions are equal whenever they are pointwise equal. It is independent from CIC, however constructing a model that negates it is not that easy, since in such a model $\Pi$-types cannot be interpreted as usual set-theoretic functions, since those are by definition extensional.

The translation is described in Figure 8 on $\mathrm{CC}_\omega$, it can be extended functorially on the identity type. Note the different steps of the translation: the translation for term $[\cdot]$ is defined inductively, using an operation (here the identity) that maps the translation $[A]$ of a type of the source to a type $[\![A]\!]$ of the target, and it is extended pointwise from types to contexts. This is the general pattern for such definitions.

$$[x] := x \qquad [\Box_i] := \Box_i \qquad [\lambda\, x : A.t] := (\lambda\, x : [\![A]\!].[t], \mathrm{true}) \qquad [t\ t'] := (\pi_1[t])\ [t']$$

$$[\Pi\, x : A.B] := (\Pi\, x : [\![A]\!].[B]) \times \mathbf{B}$$

$$[\![A]\!] := [A] \qquad [\![\cdot]\!] := \cdot \qquad [\![\Gamma, x : A]\!] := [\![\Gamma]\!], x : [\![A]\!]$$

**Figure 8:** The "$\times\mathbf{B}$" translation

This translation has the main property of a syntactic model: it preserves the typing judgement:

**Proposition 15 (Preservation of typing)**
If $\Gamma \vdash t : A$ in the source theory, then $[\![\Gamma]\!] \vdash [t] : [\![A]\!]$ in the target theory.

The proof of this proposition needs the following lemmas, which have to be true in any syntactical model:

**Lemma 16 (Preservation of substitution, conversion)**
We have $[t\{x := t'\}] \equiv [t]\{x := [t']\}$ for any terms $t$ and $t'$ of the source, and therefore if $t \equiv t'$ in the source then $[t] \equiv [t']$ in the target.

Now that we have a model of $\mathrm{CC}_\omega$, we can extend it to a model of $\mathrm{CC}_\omega$+$\neg\mathrm{funext}$, by simply inhabiting the type $[\![\neg\mathrm{funext}]\!]$. This can be done by picking $A$ and $B$ to be $\mathbf{B}$, $f$ to be $(\lambda\, x : \mathbf{B}.x, \mathrm{true})$ and $g$ to be $(\lambda\, x : \mathbf{B}.x, \mathrm{false})$. By function extensionality, $f$ and $g$ are equal, from which we get $\mathrm{true} = \mathrm{false}$, and the absurd.

## 4.2. Battle plan

Now that we have an idea of how to realize the cast operator, we can look at our battle plan, in Figure 9.

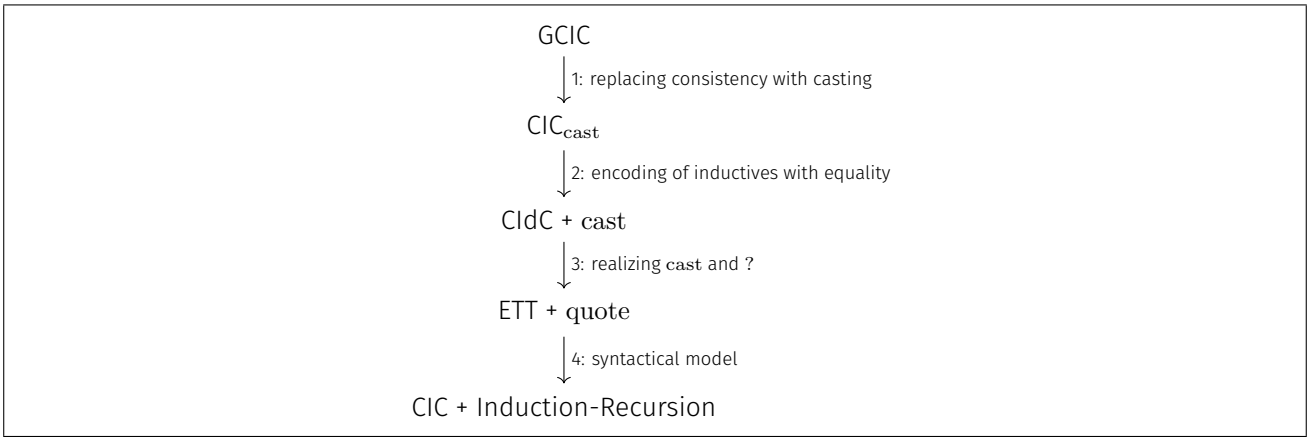


**Figure 9:** Battle plan

We already described the type theories GCIC and $\mathrm{CIC}_{\mathrm{cast}}$, as well as step 1 of the translation in the previous section. Let us detail a bit more the rest of the plan.

The theory CIdC corresponds to the theory where all inductives are indices-free, apart from the equality. We showed how to transform every inductive type in this way in Section 1.2, which corresponds to step 2.

Step 3 realizes the abstract $\mathrm{cast}$ and ? operators via a syntactical translation – this is the most novel part of this work. Instead of doing this translation directly back to CIC, we use an intermediate type theory ETT + $\mathrm{quote}$, that mixes two different constructions. The system ETT (for Exceptional Type Theory) is a version of CIC with errors, first described in [PT18] as a syntactical model. This is not the end of the work on errors in CIC, which is continued in [Péd+19]. However it is enough to serve as a base for their use within CIC, even if more developments in those line of works might give stronger properties for GCIC in the future. The $\mathrm{quote}$ operator is a type quoting operator, that allows to do recursion on types (also known as ad-hoc polymorphism). A syntactical model for this operator is given in [BPT17].

Thus step 4 mainly consists in making those two translations work together in order to realize at the same time errors and ad-hoc polymorphism. Note that the final target is not pure CIC, but also contains Induction-Recursion [Dyb00], a generalization of inductive types. This feature is necessary for the syntactical translation of ad-hoc polymorphism.

## 4.3. Realizing the Cast

TARGET LANGUAGE To realize the cast, our target language is CIC extended by some axioms:

- a function $\mathrm{raise}$, intuitively corresponding to error raising
- a family $\mathrm{quote}_i$ of quoting operators
- three functions $\mathrm{tag}$, $\mathrm{Untag}$ and $\mathrm{untag}$ to respectively construct ($\mathrm{tag}$) and destruct ($\mathrm{Untag}$ and $\mathrm{untag}$) a term of type $\mathrm{raise}\ \square$

Their typing rules are given in Figure 10. Instead of giving the exact general shape of $P_I$, we give it on our favourite examples. It can be obtained on a generic inductive using parametricity techniques, in order to exactly pin down the available inductive hypothesis, and of course there needs to be one such clause for each inductive in the source theory.

$$\vdash \mathrm{raise} : \Pi\, A : \square.A \qquad \vdash \mathrm{tag} : \Pi\, A : \square.A \to \mathrm{raise}\ \square \qquad \vdash \mathrm{Untag} : \mathrm{raise}\ \square \to \square$$

$$\vdash \mathrm{untag} : \Pi\, x : \mathrm{raise}\ \square, \mathrm{Untag}\ x \qquad \vdash \mathrm{quote}_0 : \Pi\, P : \square_0 \to \square_0.P_0^0 \to P_{\mathrm{Id}}^0 \to P_I \to P_{\mathrm{raise}}^0 \to \Pi\, A : \square_0.PA$$

$$\vdash \mathrm{quote}_{i+1} : \Pi\, P : \square_{i+1} \to \square_{i+1}.P\ \square_i \to P_0^{i+1} \to \cdots \to P_{i+1}^{i+1} \to P_{\mathrm{Id}}^{i+1} \to P_I \to P_{\mathrm{raise}}^{i+1} \to \Pi\, A : \square_{i+1}.PA$$

where

$$P_j^i := \Pi\, A : \square_j, B : (A \to \square_i).(\Pi\, x : A.P\ (B\ x)) \to P\ (\Pi\, x : A.B\ x)$$

$$P_i^i := \Pi\, A : \square_i, B : (A \to \square_i).P\ A \to (\Pi\, x : A.P\ (B\ x)) \to P\ (\Pi\, x : A.B\ x)$$

$$P_{\mathrm{Id}}^i := \Pi\, A : \square_i, \Pi\, a, a' : A.P\ (\mathrm{Id}_A\ a\ a') \qquad P_{\mathrm{raise}}^i := P\ (\mathrm{raise}\ \square_i) \qquad P_{\mathbf{N}} := P\ \mathbf{N}$$

$$P_\Sigma := \Pi\, A : \square_i.P\ A \to \Pi\, B : A \to \square_i.(\Pi\, x : A.P\ (B\ x)) \to P(\Sigma\, x : A.B\ x)$$

$$P_{\mathrm{Id}}^i := \Pi\, A : \square_i.P\ A \to \Pi\, a, a' : A.P\ (\mathrm{Id}_A\ a\ a')$$

**Figure 10:** Axioms of ETT + $\mathrm{quote}$

The functions $\mathrm{raise}$ and $\mathrm{quote}$ are not very surprising: $\mathrm{raise}$ corresponds to some kind of polymorphic error, and $\mathrm{quote}$ to induction on types – with one clause for each possible case for a type in normal form (a universe, a $\Pi$-type, an inductive, or an error).

The last three functions are much more surprising, as they really encompass the way we think of for ?. As a term, it is uninformative, but used as a type it has a different semantics: a term of type ? $\square$ can be a term of any type, because ? stands for all possible types. So to construct a term of type ? $\square$, one must provide a term $a$ together with its type $A$, which can be any type. This is what $\mathrm{tag}$ does. On the contrary, the $\mathrm{Untag}$ and $\mathrm{untag}$ primitive enable us to get back the type and term hidden in a term of type ? $\square$.

This semantics can be linked with at least two different intuitions. The first one is to remark that the primitives we provide for ? $\square$ correspond to a negative presentation of the type $\Sigma\, A : \square.A$, so ? $\square$ really corresponds to the disjoint sum of all possible types. Another way to consider it is to look into what happens

in dynamically typed languages: usually, during executions, values are tagged with their type, and every time they are used the tag is inspected to be sure the value has the correct type. We chose the names $\mathrm{tag}$ and $\mathrm{untag}$ with that comparison in mind, as this is what they do: $\mathrm{tag}$ forms a term of type $? \ \square$ by tagging a term with its type, while $\mathrm{Untag}$ and $\mathrm{untag}$ respectively get the type and term of a tagged term.

SYNTACTIC TRANSLATION Now, using those primitives, we need to give the translations $[\cdot]$ and $[\![\cdot]\!]$ from $\mathrm{CIC}_{\mathrm{cast}}$ to the target we just described. The translation of the whole CIC fragment is completely transparent: we set $[\![A]\!] := [A]$, and $[\cdot]$ is recursively defined on a term as the identity, apart from the axioms $?$ and $\mathrm{cast}$.

For those, again the case of $?$ is quite straightforward: we set $[?] := \mathrm{raise}$, because we think of the term $?$ as an error.

Now only the actually tricky point remains: defining $[\mathrm{cast}] : \Pi(A : \square), (B : \square).A \to B$. We define $[\mathrm{cast}]$ by "pattern-matching" on its first two arguments, i.e. using $\mathrm{quote}$ twice. Rather than giving an actual term using $\mathrm{quote}$, we give the intended return term with each possible pair of types, in Figure 11. We do not give $t_I$ in full generality, but only its type, intuition, and some concrete examples.

| $A$ \ $B$ | $\square_i$ | $\Pi\, y : B_1.B_2$ | $\mathrm{Id}_A\ a\ a'$ | $I'\ y_1 \dots y_n$ | $\mathrm{raise}\ \square_i$ |
|---|---|---|---|---|---|
| $\square_j$ | $\lambda\, x : \square_i.x$ if $i = j$<br>$\mathrm{raise}$ otw. | $\mathrm{raise}$ | $\mathrm{raise}$ | $\mathrm{raise}$ | $\lambda\, a : A.\mathrm{tag}\ A\ a$ |
| $\Pi\, x : A_1.A_2$ | $\mathrm{raise}$ | $t_\Pi$ | $\mathrm{raise}$ | $\mathrm{raise}$ | $\lambda\, a : A.\mathrm{tag}\ A\ a$ |
| $\mathrm{Id}_B\ b\ b'$ | $\mathrm{raise}$ | $\mathrm{raise}$ | $\mathrm{raise}$ | $\mathrm{raise}$ | $\lambda\, a : A.\mathrm{tag}\ A\ a$ |
| $I\ x_1 \dots\ x_m$ | $\mathrm{raise}$ | $\mathrm{raise}$ | $\mathrm{raise}$ | $\tilde{t_I}$ if $I = I'$<br>$\mathrm{raise}$ otw. | $\lambda\, a : A.\mathrm{tag}\ A\ a$ |
| $\mathrm{raise}\ \square_j$ | $t_{\mathrm{untag}}$ | $t_{\mathrm{untag}}$ | $t_{\mathrm{untag}}$ | $t_{\mathrm{untag}}$ | $\lambda\, x : A.x$ if $i = j$<br>$\mathrm{raise}$ otw. |

$$t_\Pi := \lambda(f : \Pi\, x : A_1.A_2), (y : B_1).\, \mathrm{cast}_{A_2\{x := \mathrm{cast}_{B_1,A_1}\ y\},B_2}(f\ \ \mathrm{cast}_{B_1,A_1}\ y)$$

$$t_{\mathrm{untag}} := \lambda\, x : \mathrm{raise}\ \square.\, \mathrm{cast}_{\mathrm{Untag}\ x,B}\ \mathrm{untag}\ x$$

$$\tilde{t_I} := t_I\ x_1 \dots x_n\ y_1 \dots y_n$$

$$t_I : \Pi(x_1 : A_1), \dots, (x_n : A_n), (x : I\ x_1 \dots\ x_n), (y_1 : A_1), \dots, (y_n : A_n\{x_i := y_i\}).I\ y_1 \dots\ y_n$$

**Figure 11:** Pattern-matching definition of $[\mathrm{cast}]\ A\ B$

Outside of the diagonal, $[\mathrm{cast}_{A,B}]$ is only defined when either $A$ or $B$ is $[?]$, otherwise it is just an error: types don't match so casting fails. When $A$ is $[?]$, untagging happens, and the value that is obtained is recursively cast using the type obtained. On the contrary, when $B$ is $[?]$, the value of type $A$ is tagged. Combined together, these two rules are quite similar to the cast cancellation rule in the Cast Calculus: a cast from $A$ to $[?]$ followed by a cast from $[?]$ to $B$ corresponds to a cast from $A$ to $B$.

On the diagonal, on the contrary, the cast should succeed. In case of a base case, i.e. a universe or an error, the cast is simply discarded. In case of a $\Pi$ a recursive call happens, again similarly to what happened for the arrow type in STLC. For inductive types as well an inductive call happens, using the term $t_I$ that basically keeps the shape of its argument intact, only inserting cast all over the place to correct the type. Examples are given in Figure 12. The case of $\mathrm{Vect}'$ is the most interesting, as the original definition used indices: note how everything happening around the indices is shifted to the cast between equality types.

Finally, note that for the identity a cast never succeeds. This is because in general there is no way to decide whether the arguments of the identity type are equal or not. However, this very pessimistic behaviour could be made a lot better by resorting a decision procedure when it exists, and considering the identity type itself as inductively defined. Those ideas are in relation with the frameworks developed in [DTT18] or [TTS18]. We do not describe this possibility in this report, at it would add another level of complexity, but we believe it would be an important piece of an actual implementation of GCIC. The impact of that systematic failure, together with the encoding of inductive types using equality, is that whenever indices are actually inspected, which amounts to destroying the equalities systematically added to the constructors, the obtained term always fails. A special case on vectors is studied in Section 5.1.

## 4.4. Realizing the Intermediate Language

The last level of the syntactic model is to give a model of ETT + $\mathrm{quote}$. It is novel, in the sense that both features have not been given a model together, however both ETT and ad-hoc polymorphism have been given a syntactic model, the first in [PT18], and the second in [BPT17].

$$t_{\mathbf{B}} := \mathrm{rec}_{\mathbf{B}}\ (\lambda\, b : \mathbf{B}.\mathbf{B})\ \mathrm{true}\ \mathrm{false} \qquad t_{\mathbf{N}} := \mathrm{rec}_{\mathbf{N}}\ (\lambda\, n : \mathbf{N}.\mathbf{N})\ 0\ (\lambda\, n : \mathbf{N}.S)$$

$$t_{\Sigma} := \lambda(A : \square), (A' : \square \to A), (s : \Sigma\, x : A.A'\ x), (B : \square), (B' : B \to \square).u_{\Sigma}$$

$$u_{\Sigma} := \mathrm{rec}_{\Sigma}\ A\ A'\ (\lambda\, s : (\Sigma\, x : A.A'\ x).\, \Sigma\, y : B.B'\ y)\ (\lambda\, a : A, a' : (A'\ a).(\mathrm{cast}_{A,B}\ a, \mathrm{cast}_{A'a,B'\ \mathrm{cast}_{A,B}\ a}\ a')\ s$$

$$t_{\mathrm{Vect}'} := \lambda(A : \square), (m : \mathbf{N}), (v : \mathrm{Vect}'\ A\ m), (B : \square), (n : \mathbf{N}).u_{\mathrm{Vect}'}$$

$$u_{\mathrm{Vect}'} := \mathrm{rec}_{\mathrm{Vect}'}\ A\ (\lambda\, m : \mathbf{N}, v : \mathrm{Vect}'\ A\ m.\, \Pi\, n : \mathbf{N}, \mathrm{Vect}'\ B\ n)\ t_{\mathrm{nil}}\ t_{\mathrm{cons}}\ m\ v$$

$$t_{\mathrm{nil}} := \lambda(m : \mathbf{N}), (e : \mathrm{Id}_{\mathbf{N}}\ m\ 0), (n : \mathbf{N}).\mathrm{nil}\ B\ \ \mathrm{cast}_{\mathrm{Id}_{\mathbf{N}}\ m\ 0,\mathrm{Id}_{\mathbf{N}}\ n\ 0}\ e$$

$$t_{\mathrm{cons}} := \lambda(m : \mathbf{N}), (m' : \mathbf{N}), (a : A), (v' : \mathrm{Vect}'\ A\ m'), (v'' : \Pi\, n : \mathbf{N}.\mathrm{Vect}'\ B\ n), (e : \mathrm{Id}_{\mathbf{N}}\ m\ (\mathrm{S}\ m')), (n : \mathbf{N}).u_{\mathrm{cons}}$$

$$u_{\mathrm{cons}} := \mathrm{cons}\ B\ n\ \ \mathrm{cast}_{A,B}\ a\ (v''\ m')\ \ \mathrm{cast}_{\mathrm{Id}_{\mathbf{N}}\ m\ (\mathrm{S}\ m'),\mathrm{Id}_{\mathbf{N}}\ n\ (\mathrm{S}\ m')}\ e$$

**Figure 12:** Examples of casting on inductive types

AD-HOC POLYMORPHISM In [BPT17], the key feature is to present universes via so-called codes: codes form a new type $\mathrm{type}_i$, that is inductively defined together with a function $\mathrm{El}_i : \mathrm{type}_i \to \square_i$ that describes how to interpret a code as a "real" type. This mutual definition is not a usual inductive one, but a recursive-inductive one. This generalization of inductive types due to Dybjer [Dyb00] allows simultaneous definition of an inductive type and of a function using this type as domain. The shape of those definitions can be found in [BPT17], we simply add a constructor $\iota$ of $\mathrm{type}_i$ for each inductive type $I$ in the source. The type of $\iota$ can be obtained by applying the $[\![\cdot]\!]$ translation of the ad-hoc polymorphism model to the type of $I$, and El is defined to be $I$ on $\iota$, as expected. This recursive-inductive $\mathrm{type}_i$ comes with a recursion principle $\mathrm{rec}_{\mathrm{type}}$.

EXCEPTIONAL TYPE THEORY In [PT18], a framework called ETT is given for errors in ETT. The authors identify three points where some freedom is left in the translation: the type $\mathbb{E}$ of errors, and the terms $\boxtimes$ and $\star$ respectively corresponding to $[\![\mathrm{raise}\ \square]\!]$ and $[\mathrm{raise}\ \square]_{\varnothing}$, where $[\cdot]_{\varnothing}$ describes how to inhabit $[\![A]\!]$ from an error. They choose to make the translation parametric in $\mathbb{E}$, but fix $\boxtimes$ and $\star$ to be degenerate. Here we want to do the contrary. We fix $\mathbb{E}$ to be a singleton type, or even better totally erase is by replacing all instances of $\mathbb{E} \to A$ by the isomorphic $A$ ($\mathbb{E}$ only appears to the left of arrows). This is because we consider our errors as uninformative. On the contrary, our setting needs $\boxtimes$ to be non-trivial, so we define it to be the following inductive:

```
Inductive ⊠ : □ :=
  | box : Π A : type.El A → ⊠
```

And set $\star := (\top, \mathrm{tt})$.

For the rest, we closely follow the paper. We first go on to define the inductive etype representing types that can raise errors:

```
Inductive etype : □ :=
  | TypeVal : Π A : type_i.El_i A → etype
  | TypeErr : etype_i
```

Using its eliminator $\mathrm{rec}_{\mathrm{etype}}$ we define two functions eEl and Err that respectively recover the type and default element from a term of type etype:

$$\mathrm{eEl} : \mathrm{etype} \to \square$$

$$\mathrm{Err} : \Pi\, A : \mathrm{etype}.\mathrm{eEl}\ A$$

Finally, for each inductive $I$ in the source, we define an inductive $I^{\bullet}$ that corresponds to $I$ with one constructor $c^{\bullet}$ for each constructor $c$ of $I$, and an extra constructor $I^{\bullet}$ representing the primitive error on type $I$.

SYNTACTIC TRANSLATION With these defined, the translation combines the ones for ad-hoc polymorphism and exceptions: they are transparent on terms, but heavily modify the interpretation of types – this is where the tricky work is. It is given in Figure 13, with $\mathcal{U}_i$ the code for the universe $\square_i$, and $\pi$ the code for the product $\Pi$.

REALIZING THE AXIOMS To finalize our translation, we still have to realize all the axioms in the translation. For raise, tag, Untag and untag, see Figure 14. Now that we have done the hard work of defining the target, the translation of the axioms are not too much work: raise is translated to Err, and tag, Untag and untag respectively amount to the constructor box, the first, and the second projection of $\boxtimes$.

$$[\Box_i] := \mathrm{TypeVal}\ \mathcal{U}_i\ \mathrm{TypeErr}_i \qquad [x] := x \qquad [\lambda\, x : A.M] := \lambda\, x : [\![A]\!].[M] \qquad [M\ N] := [M]\ [N]$$

$$[\Pi\, x : A.B] := \mathrm{TypeVal}\ (\pi\ [A]\ (\lambda\, x : [\![A]\!].[B]))\ (\lambda\, x : [\![A]\!].[B]_\varnothing)$$

$$[I] := \lambda\, x_1 : [\![A_1]\!], \ldots, x_n : [\![A_n]\!].\mathrm{TypeVal}\ (I^\bullet\ x_1 \ldots\ x_n)\ (I_\varnothing\ x_1 \ldots\ x_n) \qquad [c] := c^\bullet \qquad [A]_\varnothing := \mathrm{Err}\ [A]$$

$$[\![A]\!] := \mathrm{El}\ [A]$$

Figure 13: Exceptional ad-hoc translation

$$\begin{array}{rcl}
[\mathrm{raise}] & : & [\![\Pi\, A : \Box.A]\!] \equiv \Pi\, A : \mathrm{etype}.\mathrm{eEl}\ A \\
 & := & \mathrm{Err} \\
[\mathrm{tag}] & : & [\![\Pi\, A : \Box.A \to \mathrm{raise}\ \Box]\!] \equiv \Pi\, A : \mathrm{etype}.\mathrm{eEl}\ A \to \boxtimes \\
 & := & \mathrm{rec}_{\mathrm{etype}}\ (\lambda\, A : \mathrm{etype}.\mathrm{eEl}\ A \to \boxtimes)\ (\lambda(A : \mathrm{type}), (A_\varnothing : \mathrm{El}\ A)(a : A).\mathrm{box}\ A\ a)\ (\lambda\, x : \boxtimes.x) \\
[\mathrm{Untag}] & : & [\![\mathrm{raise}\ \Box \to \Box]\!] \equiv \boxtimes \to \mathrm{etype} \\
 & := & \mathrm{rec}_{\boxtimes}\ (\lambda\, x : \boxtimes.\mathrm{etype})\ (\lambda\, A : \mathrm{type}, a : \mathrm{El}\ A.\mathrm{TypeVal}\ A\ a) \\
[\mathrm{untag}] & : & [\![\Pi\, x : \mathrm{raise}\ \Box.\mathrm{Untag}\ x]\!] \equiv \Pi\, x : \boxtimes.\mathrm{eEl}\ (\mathrm{Untag}\ x) \\
 & := & \mathrm{rec}_{\boxtimes}\ (\lambda\, x : \boxtimes.\mathrm{eEl}\ (\mathrm{Untag}\ x))\ (\lambda\, A : \mathrm{type}, a : \mathrm{El}\ A.a)
\end{array}$$

Figure 14: Realizations of raise, tag, Untag and untag

We do not want to write down [quote] in extenso, but its definition simply combines $\mathrm{rec}_{\mathrm{etype}}$ and $\mathrm{rec}_{\mathrm{type}}$ to do type-recursion on an etype and not only on a type.

## 4.5. Properties

The first thing to prove is that our syntactical translations are correct, i.e. they both have the following properties:

**Proposition 17 (Correctness of the syntactic translations)**
If $t$, $t'$ and $T$ are terms of the source calculus, and $\Gamma$ is a context of that source calculus, then

- if $t \mapsto t'$ then $[t] \mapsto [t']$
- if $\Gamma \vdash t : T$ then $[\![\Gamma]\!] \vdash [t] : [\![T]\!]$

**Proof (Sketch)**
The first point is direct, as for both translations $\beta$-redexes are interpreted by $\beta$-redexes, and $\iota$-redexes (application of the recursor of an inductive type to a term with a constructor of that inductive in head) as the same $\iota$-redexes.

The second point amounts to verify that all inference rules for $\vdash$ in the source are derivable in the target, and indeed they are.

From now on we denote as $[\cdot]$ (resp. $[\![\cdot]\!]$) the composition of both translations on terms (resp. the composition of the term translation from $\mathsf{CIC}_{\mathrm{cast}}$ to ETT + quote with the type component of the ad-hoc exceptional translation), that give a syntactical model of $\mathsf{CIC}_{\mathrm{cast}}$.

Because axioms do not compute, the reduction of $\mathsf{CIC}_{\mathrm{cast}}$ is strictly included in the reduction of the target. However, we want our "axioms" to compute, hence the following definition.

**Definition 18 (Reduction for $\mathsf{CIC}_{\mathbf{cast}}$)**
Define the relation $t \mapsto_{\mathrm{cast}} t'$ between terms of $\mathsf{CIC}_{\mathrm{cast}}$ as the relation $[t] \mapsto [t']$ between their translations.

This is the relation on which properties similar to the "reduction" part of the theorems on GTLC should be stated and proven. We did not extensively investigate those yet, so we leave that work open.

A very strong difference with GTLC is the following proposition, direct consequence of the strong normalization of CIC with induction-recursion. We describe in Section 5.3 what the consequence for the stereotypical non-terminating term $\Omega$ in GCIC are.

**Proposition 19 (Strong normalization of $\text{CIC}_{\text{cast}}$)**
The relation $\mapsto_{\text{cast}}$ is strongly normalizing.

Concerning consistency, as it was already the case for ETT, $\text{CIC}_{\text{cast}}$ is inconsistent: one can inhabit $\bot$ using an error. However, everything is not lost, as the next proposition illustrates.

**Proposition 20 (Weak consistency)**
If $\vdash t : \bot$ in $\text{CIC}_{\text{cast}}$, then $t \equiv \text{raise } \bot$. Rephrased, the only way to inhabit $\bot$ in $\text{CIC}_{\text{cast}}$ is to use an error.

Concerning GCIC itself, depending on the consistency relation, we might get a stronger property, however we believe that concentrating on the consistency relation to avoid $\bot$ to be inhabited is not a good aim. A better way to do so should be to use the frameworks under investigation to control errors in CIC in [PT18; Péd+19], to provide us with a better way to handle and reason about errors arising from GCIC.

# 5. Use Examples

## 5.1. Vectors

A standard example for the use of indices is the type of vectors. On the upside, they enable type-safe functions, for instance the head function of type $\Pi(A : \square), (n : \mathbf{N}).\text{Vect } A \ (\text{S } n) \to A$. On the downside, programming with vectors easily gets quite involved, for instance giving the type $\Pi(A : \square), (n : \mathbf{N}).\text{Vect } A \ n \to \text{Vect } A \ n$ to a quicksort function as the following OCaml one (inspired from an example of [ETG19])

```
let rec quicksort : int list → int list = fun l →
match l with
| [] → []
| h :: t → (quicksort (filter (fun a → a ≤ h) t))@[h]@(quicksort (filter (fun
    a → h < a) t))
```

is already involved, as it requires a subtle handling of the indices. Instead, along the gradual typing philosophy, a first step would be to give it the type $\Pi(A : \square), (n : \mathbf{N}).\text{Vect } A \ n \to \text{Vect } A \ ?$ to enable an easy definition, leaving the work of giving it a static type to a second pass.

Such a definition would insert some casts from a static $\text{Vect } A \ n$ to $\text{Vect } A \ ?$ and in the other direction as well. What happens when vectors obtained with such casts are used? As we encode indices via equality, the equality arguments are where the interesting stuff happens. For instance, let $v$ be a vector of size 3 and consider the following term

$$v' := \text{cast}_{\text{Vect } A \ 5, \text{Vect } A \ ?} \ \text{cast}_{\text{Vect } A \ ?, \text{Vect } A \ 3} \ v$$

It reduces to an actual vector of length 3 (in the sense that it uses cons three times), but with a type $\text{Vect } A \ 5$, obtained using a proof of $3 = 5$, that of course is an error. Thus, as long as $v'$ is used without inspecting that proof – in particular as long as $v'$ is used as a list – no error occurs. The faulty proof is simply carried around. However, if a function making actual use of the index is called, it exposes the lie, and raises an error. For instance, if one were to call a function taking the $4^{\text{th}}$ element of a list of length at least 4 on $v'$, similar to the safe head function, one would end up in a branch that is supposedly unreachable because of typing, and where a term of type $A$ is obtained by elimination of $\bot$. Here this branch would actually be reached, but the error would propagate through the proof of false and its elimination, so that the provided term of type $A$ would simply be $\text{raise } A$.

In general, indices are used to reason in unreachable branches, where they are used to create a proof of $\bot$ that can be eliminated. With gradual types, those branches might actually be reached using erroneous proofs, and so they would simply return an error. On the contrary indices are not useful in reachable branches, so as long as they are not needlessly inspected there, no error should be raised.

## 5.2. $\eta$-rule for inductive types

The need to have a return predicate when destroying terms of an inductive types (the first argument $P$ of all recursors) is quite an annoying feature in practice, as writing that return predicate explicitly is tedious. A desirable possibility is to get rid of that predicate in a systematic way. For booleans, this would give a recursor of the following type

$$\mathrm{rec}'_{\mathbf{B}} : \Pi(P_1 : \Box)(P_2 : \Box).P_1 \to P_2 \to \Pi\, b : \mathbf{B}.\mathrm{rec}'_{\mathbf{B}}\ \Box\ \Box\ P_1\ P_2\ b$$

In a proof assistant, the types $P_1$ and $P_2$ can be inferred, alleviating the user from the burden of giving an explicit $P$. However, because $\mathrm{rec}'_{\mathbf{B}}$ is typed using itself, such a recursor cannot be used, unless the theory features so-called $\eta$-conversion, that allows the following reduction: $\mathrm{rec}'_{\mathbf{B}}\ P\ P\ p\ p\ b \mapsto_\eta p$. Indeed, using that rule, in the previous example $\mathrm{rec}'_{\mathbf{B}}\ \Box\ \Box\ P_1\ P_2\ b$ can be given type $\Box$ instead of $\mathrm{rec}'_{\mathbf{B}}\ \Box\ \Box\ \Box\ \Box\ b$.

Sadly, $\eta$-reduction for inductive types cannot be part of the theory, for decidability reasons: already on natural numbers it is undecidable to know whether $\eta$-reduction applies.

However, in a gradual setting, $\eta$-conversion on types can be simulated: as soon as we have

$$\mathrm{rec}'_{\mathbf{B}}\ \Box\ \Box\ P\ P\ b \sim P$$

a term of the first type can be used as a term of the second, as if $\eta$-conversion had happened. Whenever the underlying term $\iota$-reduces because $b$ has taken a concrete value, the $\mathrm{cast}$ inserted around it reduces to $\mathrm{cast}_{P,P}$, and if $P$ is simple enough (as in our example) it completely disappears.

## 5.3. Non-terminating terms actually terminate

A very interesting discovery we made while investigating GCIC concern the term $\Omega$ defined as follows:

$$\Omega := (\lambda\, x : ?\, .x\ x)\ (\lambda\, x : ?\, .x\ x)$$

In pure lambda calculus, $\Omega$ is the stereotypical non-normalizable term. In most gradually typed languages, for instance GTLC, such a term is typable, and it prevents the system from being strongly normalizing. In GCIC, in contrast, a particularly peculiar thing happens: $\Omega$ is typable, however the type hierarchy prevents it from looping. Instead, $\Omega$ reduces to an error. The detailed reduction is presented in the Appendix E, the key point is that to make $\Omega$ loop a cast from $?\ \Box_{i+1}$ to $?\ \Box_i$ is needed, and as this is not the identity but an error, the whole term fails.

# Conclusion and Future Work

In this report, we presented a novel approach to gradualization in the context of CIC, the first one to try and tame the whole complexity of CIC. Its main theoretical contribution is the casting operator, presented through a syntactical translation, using the underspecification of the framework of ETT to give the error the semantics of a gradual type. Even if the intuitions behind this idea are not new, their use to interpret dynamic types as an inductive had not been observed before.

Although the structure of GCIC is clear, there still remains some work to do on exact details pertaining to universe levels and full general definitions for inductive types. Another line of work we did not explore yet is to implement this work in Coq, using the meta-programming features of the TemplateCoq/MetaCoq project [Ana+18] and drawing inspiration from the plugins for program translation of [BPT17; PT18]. Both can go hand in hand, using the automated handling of universes in Coq to alleviate some of the burden of doing all of it by hand, an approach we already successfully used to back up our insight on the terminating $\Omega$, or construct and check some of quite indigestible terms, e.g. Figure 12.

# References


[Ana+18] A. Anand et al. “Towards Certified Meta-Programming with Typed Template-Coq”. In: *ITP 2018 - 9th Conference on Interactive Theorem Proving*. Vol. 10895. LNCS. Oxford, United Kingdom: Springer, July 2018, pp. 20–39. DOI: 10.1007/978-3-319-94821-8_2.

[Asp+12] A. Asperti et al. “A Bi-Directional Refinement Algorithm for the Calculus of (Co)Inductive Constructions”. In: *Logical Methods in Computer Science* Volume 8, Issue 1 (Mar. 2012). DOI: 10.2168/LMCS-8(1:18)2012. URL: https://lmcs.episciences.org/1044.

[Bar91] H. Barendregt. “An Introduction to Generalized Type Systems”. In: *Journal of Functional Programming* 1 (Apr. 1991), pp. 125–154. DOI: 10.1017/S0956796800020025.

[BPT17] S. Boulier, P.-M. Pédrot, and N. Tabareau. “The next 700 syntactical models of type theory”. In: *Certified Programs and Proofs (CPP 2017)*. Paris, France, Jan. 2017, pp. 182–194. DOI: 10.1145/3018610.3018620. URL: https://hal.inria.fr/hal-01445835.

[CH88] T. Coquand and G. Huet. “The calculus of constructions”. In: *Information and Computation* 76.2 (1988), pp. 95–120. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(88)90005-3.

[Dow01] G. Dowek. “Chapter 16 - Higher-Order Unification and Matching”. In: *Handbook of Automated Reasoning*. Ed. by A. Robinson and A. Voronkov. Handbook of Automated Reasoning. Amsterdam: North-Holland, 2001, pp. 1009–1062. ISBN: 978-0-444-50813-3. DOI: https://doi.org/10.1016/B978-044450813-3/50018-7. URL: http://www.sciencedirect.com/science/article/pii/B9780444508133500187.

[DTT18] P.-É. Dagand, N. Tabareau, and É. Tanter. “Foundations of Dependent Interoperability”. In: *Journal of Functional Programming* 28 (2018), 9:1–9:44.

[Dyb00] P. Dybjer. “A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory”. In: *Journal of Symbolic Logic* 65 (June 2000). DOI: 10.2307/2586554.

[ETG19] J. Eremondi, É. Tanter, and R. Garcia. “Approximate Normalization for Gradual Dependent Types”. In: *Proceedings of the ACM on Programming Languages* 3.ICFP (Aug. 2019).

[GCT16] R. Garcia, A. M. Clark, and É. Tanter. “Abstracting Gradual Typing”. In: *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL 2016)*. St Petersburg, FL, USA: ACM Press, Jan. 2016, pp. 429–442.

[Gri90] T. G. Griffin. “A Formulae-as-type Notion of Control”. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’90. ACM, 1990, pp. 47–58. DOI: 10.1145/96709.96714.

[LT17] N. Lehmann and É. Tanter. “Gradual Refinement Types”. In: *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*. Paris, France: ACM Press, Jan. 2017, pp. 775–788.

[Pau15] C. Paulin-Mohring. “Introduction to the Calculus of Inductive Constructions”. In: *All about Proofs, Proofs for All*. Ed. by B. W. Paleo and D. Delahaye. Vol. 55. Studies in Logic (Mathematical logic and foundations). College Publications, 2015.

[Pau93] C. Paulin-Mohring. “Inductive Definitions in the System Coq - Rules and Properties”. In: *Proceedings of the conference Typed Lambda Calculi and Applications*. Ed. by M. Bezem and J.-F. Groote. Lecture Notes in Computer Science 664. LIP research report 92-49. 1993.

[Péd+19] P.-M. Pédrot et al. “A Reasonably Exceptional Type Theory”. In: *ICFP 2019 - 24th ACM SIGPLAN International Conference on Functional Programming*. Berlin, Germany: ACM, Aug. 2019. DOI: 10.1145/3341712.

[PT18] P.-M. Pédrot and N. Tabareau. “Failure is Not an Option An Exceptional Type Theory”. In: *ESOP 2018 - 27th European Symposium on Programming*. Vol. 10801. LNCS. Thessaloniki, Greece: Springer, 2018, pp. 245–271. DOI: 10.1007/978-3-319-89884-1_9.

[RLL14] G. van Rossum, J. Lehtosalo, and Ł. Langa. *PEP 484 – Type Hints*. Tech. rep. The Python Software Foundation, 2014. URL: https://www.python.org/dev/peps/pep-0484/.

[Sie+15] J. G. Siek et al. “Refined Criteria for Gradual Typing”. In: *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Ed. by T. Ball et al. Vol. 32. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 274–293. DOI: 10.4230/LIPIcs.SNAPL.2015.274.

[The19] The Coq Development Team. *The Coq Proof Assistant, version 8.9.0*. Jan. 2019. DOI: 10.5281/zenodo.2554024.

[TLT19] M. Toro, E. Labrada, and É. Tanter. "Gradual Parametricity, Revisited". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 17:1–17:30. DOI: 10.1145/3290330.

[TT15] É. Tanter and N. Tabareau. "Gradual Certified Programming in Coq". In: *Proceedings of the 11th ACM Dynamic Languages Symposium (DLS 2015)*. Pittsburgh, PA, USA: ACM Press, Oct. 2015, pp. 26–40.

[TTS18] N. Tabareau, É. Tanter, and M. Sozeau. "Equivalences for Free". In: *Proceedings of the ACM on Programming Languages*. ICFP'18 2.ICFP (Sept. 2018), pp. 1–29. DOI: 10.1145/3234615. URL: https://hal.inria.fr/hal-01559073.

[ZS17] B. Ziliani and M. Sozeau. "A comprehensible guide to a new unifier for CIC including universe polymorphism and overloading". In: *Journal of Functional Programming* 27 (2017), e10. DOI: 10.1017/S0956796817000028.

# Appendices

## A. Usual Inductive Types

Booleans:

```
Inductive B : □ :=
| true : B
| false : B
```

$$\mathrm{rec}_{\mathbf{B}} : \Pi\, P : \mathbf{B} \to \square.(P\ \mathrm{true}) \to (P\ \mathrm{false}) \to \Pi\, b : \mathbf{B}.P\ b$$

$$\mathrm{rec}_{\mathbf{B}}\ P\ t_{\mathrm{true}}\ t_{\mathrm{false}}\ \mathrm{true} \mapsto_{\iota} t_{\mathrm{true}}$$

$$\mathrm{rec}_{\mathbf{B}}\ P\ t_{\mathrm{true}}\ t_{\mathrm{false}}\ \mathrm{false} \mapsto_{\iota} t_{\mathrm{false}}$$

Natural numbers:

```
Inductive N : □ :=
| 0 : N
| S : N → N
```

$$\mathrm{rec}_{\mathbf{N}} : \Pi\, P : \mathbf{N} \to \square.(P\ 0) \to (\Pi\, n : \mathbf{N}.(P\ n) \to (P\ (\mathrm{S}\ n)) \to \Pi\, n : \mathbf{N}.P\ n$$

$$\mathrm{rec}_{\mathbf{N}}\ P\ t_0\ t_{\mathrm{S}}\ 0 \mapsto_{\iota} t_0$$

$$\mathrm{rec}_{\mathbf{N}}\ P\ t_0\ t_{\mathrm{S}}\ (\mathrm{S}\ t) \mapsto_{\iota} t_{\mathrm{S}}\ t\ (\mathrm{rec}_{\mathbf{N}}\ P\ t_0\ t_{\mathrm{S}}\ t)$$

Dependant sum:

```
Inductive Σ (A : □) (B : A → □) : □ :=
| p : Π a : A, b : B a. Σ A B
```

$$\mathrm{rec}_{\Sigma} : \Pi\, A : \square, B : A \to \square, P : (\Sigma\ A\ B) \to \square.(\Pi\, a : A, b : B\ a.P\ (a, b)) \to \Pi\, s : (\Sigma\ A\ B).P\ s$$

$$\mathrm{rec}_{\Sigma}\ A\ B\ P\ t_{\mathrm{p}}\ (a, b) \mapsto_{\iota} t_{\mathrm{p}}\ a\ b$$

Equality:

```
Inductive Id (A : □) (a : A) (a' : A) : □ :=
| refl : Id A a a
```

$$\mathrm{rec}_{\mathrm{Id}} : \Pi\, A : \square, a : A, P : (\Pi\, a' : A.(\mathrm{Id}\ A\ a\ a') \to \square).(P\ a\ (\mathrm{refl}\ A\ a)) \to \Pi\, e : \mathrm{Id}\ A\ a\ a'.P\ a'\ e$$

$$\mathrm{rec}_{\mathrm{Id}}\ A\ a\ P\ t_{\mathrm{refl}}\ (\mathrm{refl}\ A\ a) \mapsto_{\iota} t_{\mathrm{refl}}$$

Vectors:

```
Inductive Vect (A : □) (n : N) : □ :=
| nil : Vect A 0
| cons : Π n : N, a : A, v : Vect A n.Vect A (S n)
```

$$\mathrm{rec}_{\mathrm{Vect}} : \Pi\, A : \square, P : (\Pi\, n : \mathbf{N}.\mathrm{Vect}\ A\ n \to \square).(P\ 0\ (\mathrm{nil}\ A)) \to$$

$$(\Pi\, n : \mathbf{N}, a : A, v : \mathrm{Vect}\ A\ n.P\ n\ v \to P\ (\mathrm{S}\ n)\ (\mathrm{cons}\ A\ n\ a\ v)) \to \Pi\, n : \mathbf{N}, v : \mathrm{Vect}\ A\ n.P\ n\ v$$

$$\mathrm{rec}_{\mathrm{Vect}}\ A\ P\ t_{\mathrm{nil}}\ t_{\mathrm{cons}}\ (\mathrm{nil}\ A) \mapsto_{\iota} t_{\mathrm{nil}}$$

$$\mathrm{rec}_{\mathrm{Vect}}\ A\ P\ t_{\mathrm{nil}}\ t_{\mathrm{cons}}\ (\mathrm{cons}\ A\ n\ a\ v) \mapsto_{\iota} t_{\mathrm{cons}}\ n\ a\ v\ (\mathrm{rec}_{\mathrm{Vect}}\ A\ P\ t_{\mathrm{nil}}\ t_{\mathrm{cons}}\ v)$$

Unit:

```
Inductive ⊤ : □ :=
| tt : ⊤
```

$$\mathrm{rec}_\top : \Pi\, P : \top \to \square, P\ \mathrm{tt} \to \Pi\, x : \top . P\ x$$

$$\mathrm{rec}_\top\ P\ p\ \mathrm{tt} \mapsto_\iota p$$

False:

```
Inductive ⊥ : □ :=
```

$$\mathrm{rec}_\bot : \Pi\, P : \bot \to \square, x : \bot . P\ x$$

## B. Equality Encoding of Vectors

```
Inductive Vect′ (A : □) (n : N) : □ :=
| nil′ : Id_N n 0 → Vect′ A n
| cons′ : Π m : N, a : A, v : Vect A m.(Id_N S m n) → Vect A n
```

$$\mathrm{nil}'' := \lambda\, A : \square . \mathrm{nil}'\ A\ 0\ (\mathrm{refl}\ \mathbf{N}\ 0)$$

$$\mathrm{cons}'' := \lambda\, A : \square, n : \mathbf{N}, a : A, v : \mathrm{Vect}'\ A\ n . \mathrm{cons}'\ A\ (\mathrm{S}\ n)\ n\ a\ v\ (\mathrm{refl}\ \mathbf{N}\ (\mathrm{S}\ n))$$

$$\vdash \mathrm{rec}'_{\mathrm{Vect}} : \Pi\, A : \square, P : (\Pi\, n : \mathbf{N} . \mathrm{Vect}'\ A\ n \to \square) . P\ 0\ (\mathrm{nil}''\ A) \to$$

$$(\Pi\, n : \mathbf{N}, a : A, v : \mathrm{Vect}'\ A\ n . P\ n\ v \to P\ (\mathrm{S}\ n)(\mathrm{cons}''\ A\ n\ a\ v)) \to \Pi\, n : \mathbf{N}, v : \mathrm{Vect}'\ A\ n . P\ n\ v$$

## C. Cast Calculus

### C.1. Typing Rules

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{}{\Gamma \vdash \underline{n} : \mathbf{N}} \qquad \frac{}{\Gamma \vdash \underline{b} : \mathbf{B}} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda\, x : T_1 . t : T_1 \to T_2}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2 \qquad \mathrm{dom}(T_1) = T_2}{\Gamma \vdash t_1\ t_2 : \mathrm{cod}(T_2)} \qquad \frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2 \qquad T_1 = \mathbf{N} \qquad T_2 = \mathbf{N}}{\Gamma \vdash t_1 + t_2 : \mathbf{N}}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2 \qquad \Gamma \vdash t_3 : T_3 \qquad T_1 = \mathbf{B} \qquad T_2 = T \qquad T_3 = T}{\Gamma \vdash \mathrm{if}\ t_1\ \mathrm{then}\ t_2\ \mathrm{else}\ t_3 : T} \qquad \frac{}{\Gamma \vdash \mathrm{raise} : T}$$

$$\frac{\Gamma \vdash t : T_1}{\Gamma \vdash \mathrm{cast}_{T_1, T_2}\ t : T_2}$$

### C.2. Compilation Rules

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x \rightsquigarrow x : T} \qquad \frac{}{\Gamma \vdash \underline{n} \rightsquigarrow \underline{n} : \mathbf{N}} \qquad \frac{}{\Gamma \vdash \underline{b} \rightsquigarrow \underline{b} : \mathbf{B}} \qquad \frac{\Gamma, x : T_1 \vdash t \rightsquigarrow t' : T_2}{\Gamma \vdash \lambda\, x : T_1 . t \rightsquigarrow \lambda\, x : T_1 . t' : T_1 \to T_2}$$

$$\frac{\Gamma \vdash t_1 \rightsquigarrow t'_1 : T_1 \qquad \Gamma \vdash t_2 \rightsquigarrow t'_2 : T_2 \qquad \mathrm{dom}_?(T_1) \sim T_2}{\Gamma \vdash t_1\ t_2 \rightsquigarrow t'_1\ (\mathrm{cast}_{T_2, \mathrm{dom}_?(T_1)}\ t'_2) : \mathrm{cod}_?(T_1)}$$

$$\frac{\Gamma \vdash t_1 \rightsquigarrow t'_1 : T_1 \qquad \Gamma \vdash t_2 \rightsquigarrow t'_2 : T_2 \qquad T_1 \sim \mathbf{N} \qquad T_2 \sim \mathbf{N}}{\Gamma \vdash t_1 + t_2 \rightsquigarrow \mathrm{cast}_{T_1, \mathbf{N}}\ t'_1 + \mathrm{cast}_{T_2, \mathbf{N}}\ t'_2 : \mathbf{N}}$$

$$\frac{\Gamma \vdash t_1 \rightsquigarrow t'_1 : T_1 \qquad \Gamma \vdash t_2 \rightsquigarrow t'_2 : T_2 \qquad \Gamma \vdash t_3 \rightsquigarrow t'_3 : T_3 \qquad T_1 \sim \mathbf{B} \qquad T_2 \sim T \qquad T_3 \sim T}{\Gamma \vdash \mathrm{if}\ t_1\ \mathrm{then}\ t_2\ \mathrm{else}\ t_3 \rightsquigarrow \mathrm{if}(\mathrm{cast}_{T_1, \mathbf{B}}\ t'_1)\ \mathrm{then}(\mathrm{cast}_{T_2, T}\ t'_2)\ \mathrm{else}(\mathrm{cast}_{T_3, T}\ t'_3) : T}$$

## D. Compilation rules for GCIC

$$\frac{}{\vdash \cdot} \qquad \frac{\Gamma \vdash A : \square \rightsquigarrow A'}{\vdash \Gamma, x : A} \qquad \frac{\Gamma \vdash B : \square \rightsquigarrow B' \qquad \Gamma \vdash x : A \rightsquigarrow t'}{\Gamma, y : B \vdash x : A \rightsquigarrow t'} \qquad \frac{\vdash \Gamma, x : A}{\Gamma, x : A \vdash x : A \rightsquigarrow x}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \square_i : \square_{i+1} \rightsquigarrow \square_i} \qquad \frac{\Gamma, x : A \vdash t : B \rightsquigarrow t' \qquad \Gamma \vdash \Pi\, x : A.B : \square \rightsquigarrow \Pi\, x : A'.B'}{\Gamma \vdash \lambda\, x : A.t : \Pi\, x : A.B \rightsquigarrow \lambda\, x : A'.t'}$$

$$\frac{\Gamma, x : A \vdash B : \square_i \rightsquigarrow B' \qquad \Gamma \vdash A : \square_j \rightsquigarrow A'}{\Gamma \vdash \Pi\, x : A.B : \square_{\max(i,j)} \rightsquigarrow \Pi x : A'.B'} \qquad \frac{\Gamma \vdash t : \Pi\, x : A.B \rightsquigarrow t' \qquad \Gamma \vdash u : A \rightsquigarrow u'}{\Gamma \vdash t\ u : B\{x := u\} \rightsquigarrow t'\ u'}$$

$$\frac{\Gamma \vdash T : \square \rightsquigarrow T'}{\Gamma \vdash ? : T \rightsquigarrow ?\ T'} \qquad \frac{\Gamma \vdash t : A \rightsquigarrow t' \qquad A \sim B \qquad \Gamma \vdash A : \square \rightsquigarrow A' \qquad \Gamma \vdash B : \square \rightsquigarrow B'}{\Gamma \vdash \mathrm{cast}_{A',B'}\, t'}$$

## E. Reduction of $\Omega$

We write $?_i$ for $?\ \square_i$. We first get

$$\vdash \Omega : ?_{i+1} \rightsquigarrow (\lambda\, x : ?_{i+1}\,.(\mathrm{cast}_{?_{i+1},?_{i+1}\to?_{i+1}}\ x)\ x)\ (\mathrm{cast}_{?_i\to?_i,?_{i+1}}(\lambda\, x : ?_i\,.\,\mathrm{cast}_{?_i,?_i\to?_i}\ x\ x))$$

For readability, we set

$$\delta_i := \lambda\, x : ?_i\,.(\mathrm{cast}_{?_i,?_i\to?_i}\ x)\ x$$

and (with a slight notational abuse)

$$\Omega_i := \delta_{i+1}\ (\mathrm{cast}_{?_i\to?_i,?_{i+1}}\ \delta_i)$$

The reduction now gives

$$\begin{array}{lll}
\Omega_i & \mapsto^* & (\mathrm{cast}_{?_{i+1},?_{i+1}\to?_{i+1}}\ \mathrm{cast}_{?_i\to?_i,?_{i+1}}\ \delta_i)\ (\mathrm{cast}_{?_i\to?_i,?_{i+1}}\ \delta_i) \\
& \mapsto^* & (\mathrm{cast}_{?_i\to?_i,?_{i+1}\to?_{i+1}}\ \delta_i)\ (\mathrm{cast}_{?_i\to?_i,?_{i+1}}\ \delta_i) \\
& \mapsto^* & (\lambda\, x : ?_{i+1}\,.\,\mathrm{cast}_{?_i,?_{i+1}}((\mathrm{cast}_{?_i,?_i\to?_i}\ \mathrm{cast}_{?_{i+1},?_i}\ x)\ \ \mathrm{cast}_{?_{i+1},?_i}\ x))(\mathrm{cast}_{?_i\to?_i,?_{i+1}}\ \delta_i) \\
& \mapsto^* & (\lambda\, x : ?_{i+1}\,.\,\mathrm{cast}_{?_i,?_{i+1}}((\mathrm{cast}_{?_i,?_i\to?_i}\ ?\ ?_i)\ (?\ ?_i)))(\mathrm{cast}_{?_i\to?_i,?_{i+1}}\ \delta_i) \\
& \mapsto^* & ?\ (?_{i+1})
\end{array}$$