The Multiverse: Logical Modularity for Proof Assistants

Kenji Maillard, Nicolas Margulies, Matthieu Sozeau, Nicolas Tabareau, Éric Tanter

The Multiverse: Logical Modularity for Proof Assistants

KENJI MAILLARD, Gallinette Project-Team, Inria, France
NICOLAS MARGULIES, ENS Paris-Saclay & Gallinette Project-Team, Inria, France
MATTHIEU SOZEAU, Gallinette Project-Team, Inria, France
NICOLAS TABAREAU, Gallinette Project-Team, Inria, France
ÉRIC TANTER, PLEIAD Lab, Computer Science Department (DCC), University of Chile, Chile

Proof assistants play a dual role as programming languages and logical systems. As programming languages, proof assistants offer standard modularity mechanisms such as first-class functions, type polymorphism and modules. As logical systems, however, modularity is lacking, and understandably so: incompatible reasoning principles—such as univalence and uniqueness of identity proofs—can indirectly lead to logical inconsistency when used in a given development, even when they appear to be confined to different modules. The lack of logical modularity in proof assistants also hinders the adoption of richer programming constructs, such as effects. We propose the multiverse, a general type-theoretic approach to endow proof assistants with logical modularity. The multiverse consists of multiple universe hierarchies that statically describe the reasoning principles and effects available to define a term at a given type. We identify sufficient conditions for this structuring to modularly ensure that incompatible principles do not interfere, and to locally restrict the power of dependent elimination when necessary. This extensible approach generalizes the ad-hoc treatment of the sort of propositions in the Coq proof assistant. We illustrate the power of the multiverse by describing the inclusion of Coq-style propositions, the strict propositions of Gilbert et al., the exceptional type theory of Pédrot and Tabareau, and general axiomatic extensions of the logic.

1 INTRODUCTION

Modularity is key to scalable software development [Parnas 1972]. As the adoption of proof assistants to write certified programs [Chlipala 2013] increases, software engineering aspects become crucial. Proof assistants are peculiar in that respect due to their dual role for programming and proving. Indeed, while proof assistants such as Coq [The Coq Development Team 2020] and Agda [Norell 2009] offer traditional mechanisms for modular programming, including functional abstraction and modules, they lack modularity at the logical level. To illustrate, consider two logical principles which are known to be incompatible: *univalence*, a principle coming from Homotopy Type Theory [Univalent Foundations Program 2013], which provides a rich, computationally-relevant content to equality, and *uniqueness of identity proofs* (UIP), which considers two proofs of the same equality as necessarily equal. From both principles, one can derive a contradiction. But this conflict can be quite pernicious, as inconsistency can arise from seemingly harmless consequences of these principles. For instance, using univalence one can prove that there exists an equality on the type of booleans, such that transporting `false` along this equality (noted $e\#\texttt{true}$ below) gives `false`:

$$\exists\, e : \mathbb{B} = \mathbb{B}, e\#\texttt{true} = \texttt{false}.$$

This property does not mention univalence explicitly (in its "interface"), yet combining it with UIP yields to inconsistency. In practice, this means that developers must make *global* commitments to certain reasoning principles in order to be sure that the underlying logic of their development is consistent. For instance, an Agda development that imports univalence with the pragma {-#

Authors' addresses: Kenji Maillard, Gallinette Project-Team, Inria, Nantes, France; Nicolas Margulies, ENS Paris-Saclay & Gallinette Project-Team, Inria, Nantes, France; Matthieu Sozeau, Gallinette Project-Team, Inria, Nantes, France; Nicolas Tabareau, Gallinette Project-Team, Inria, Nantes, France; Éric Tanter, PLEIAD Lab, Computer Science Department (DCC), University of Chile, Santiago, Chile.

. XXXX-XXXX/2021/1-ART1
https://doi.org/

`OPTIONS ----cubical #-}` [Vezzosi et al. 2019] must be devoted to univalence and cannot be mixed with other incompatible extensions. Likewise, concerns about whether the use of classical principles is accepted or not must be made globally. Therefore providing logical modularity is a whole new challenge in itself, not addressed by well-known modular programming constructs. The Curry-Howard correspondence between both words has its limits, unfortunately.

Another major consequence of the lack of logical modularity in proof assistants is that the use of *effects* in the programming language is typically demonized. However, software developers are well acquainted with the use of effects such as mutable state, exceptions, or control operators, and a great deal of effort involves addressing the mismatch between the pure world of proof assistants and real-world programs. Intuitively, the problem is that effects break a number of standard reasoning principles; for instance, the commutativity of addition on natural numbers is easy to prove by induction, but this induction principle is no longer valid in its full generality with effects, due to the relevance of evaluation order. Recently, Pédrot and Tabareau [2020] proved that mixing general induction principles, a.k.a. *dependent elimination*, substitution, and effects, leads to inconsistency. This incompatibility has a longer history, of course. The addition of effects to a logical system can be traced back to double-negation translations [Glivenko 1929], although the modern standpoint can be attributed to Moggi [1991], as used for instance in F⋆ [Swamy et al. 2013]. However, Barthe and Uustalu [2002] show that defining a typed CPS translation preserving dependent elimination is out of reach, and similarly, Herbelin [2005] proves that the theory behind the Coq proof assistant is inconsistent with computational classical logic under the guise of a `call/cc` operator. In retrospect, this incompatibility is an illustration of a very ancient issue: mixing computational classical logic with the axiom of choice, whose intuitionistic version is a consequence of dependent elimination, is a well-known source of foundational problems [Martin-Löf 2006].

Is all hope lost for logical modularity in proof assistants? Is there a way to encapsulate the use of effects to well-defined parts of a development so that they do not globally break logical consistency? We answer these questions affirmatively, building upon a couple of approaches developed in specific settings. First, in order to address the issue of combining univalence and UIP, Voevodsky [2013] proposed the notion of homotopy type system—later revisited as two-level type theory by Altenkirch et al. [2016]—which introduces two universe hierarchies in order to distinguish between so-called univalent types and strict types. Second, Pédrot et al. [2019] also propose the use of different universe hierarchies to support consistent reasoning about effectful programs written in the Exceptional Type Theory [Pédrot and Tabareau 2018]. Third, one can understand the well-known Type/Prop distinction in Coq under the same light: Prop—which is a one-level hierarchy indeed—lives "apart" from the Type hierarchy, with a restricted elimination schema from Prop into Type, known as *singleton elimination*. This restriction ensures that Prop is compatible with proof irrelevance—a property assumed by the extraction mechanism [Letouzey 2004] to ensure computability of erased code—because otherwise one could prove that the type of booleans in Prop has two distinct inhabitants.

While all these theories share the same substrate—Martin-Löf Type Theory [Martin-Löf 1975]—they come with their own pecularities and metatheoretical justifications, either developed on paper [Altenkirch et al. 2016; Gratzer et al. 2020; Pédrot et al. 2019; Pédrot and Tabareau 2018] or mechanized [Abel et al. 2018; Sozeau et al. 2020], involving a great deal of human effort and repeated work. Here, we develop a generic framework for defining, studying and combining such theories. For example, the addition of inductive types to a specific sort can in many cases be performed in a uniform manner. We additionally build a generic logical relation model that can accommodate multiple sorts, each with different sets of logical and computational principles, generalizing prior work by Abel et al. [2018]. To achieve this, we abstract the introduction of type constructors in a given sort and their inhabitants, along with their associated computational principles.

Contributions. Building upon this analysis and generalizing the idea of using a separate universe hierarchy to isolate a given reasoning principle or effect, this work develops the notion of the *multiverse* as a principled type-theoretic approach to endow proof assistants with logical modularity. The multiverse is a system with multiple universe hierarchies that statically describe which principles and effects are available to define a term at a given type. The multiverse permits the controlled use of incompatible reasoning principles in a development, where such principles can be used separately to establish different results about the same object of study, without any risk of unintended interference. Likewise, the multiverse makes it possible to extend the programming language of a proof assistant with effects, by locally restricting the power of dependent elimination in accordance with the considered effects. Specifically:

- We introduce MuTT, a dependent type theory parametrized by a description of multiple universes hierarchies and computational principles in §2.
- We illustrate the expressivity of the framework in §3 with a presentation of inductive types, concrete instances providing Coq-style propositions, the Exceptional Type Theory as well as general axiomatic extensions of the logic, and identify sufficient conditions for a universe to admit dependent elimination.
- We show in §4 that MuTT indeed provides a modular framework: two independent parametrizations of MuTT can be combined without endangering the metatheoretical properties of its core.
- We prove important metatheoretical results on MuTT that ensure consistency, canonicity and decidability of typechecking for any valid parametrization of the theory, showing that MuTT is suitable as an idealized theory for proof assistants implementations (§5).
- We briefly explain how the addition of extensionality principles fit in our framework and use it to describe an instance of MuTT with strict propositions (SProp) [Gilbert et al. 2019] as can be found in Coq and Agda, hence subsuming the theory of existing proof assistants.

Finally, §7 discusses related work and §8 concludes.

2 MUTT: MULTIVERSE TYPE THEORY

After a brief introduction to type theory, we present the syntax and typing of the Multiverse Type Theory (MuTT), highlighting its parametrization, along with the expected conditions that a specific MuTT parametrization must satisfy in order to be valid. Valid parametrizations of MuTT yield a type theory suitable to serve as the basis for proof assistants.

Background. Martin-Löf Type Theory (MLTT) [Martin-Löf 1971] is a dependent type theory featuring dependent products (functions), dependent sums (pairs) and identity types (equality). In MLTT there is a single *sort* for all types, which is left implicit. The sort is represented by a *universe* constructor $\square$ that classifies all types, including itself (represented as `Set` in Agda). For example, a dependent function has a product type, and that product type itself has the type $\square$. More precisely, because the sort is *predicative*, it is structured as a stratified hierarchy of universes $\square_i$, each at universe level i, so that $\square_i$ has type $\square_{i+1}$.

The Calculus of Inductive Constructions (CIC) [Paulin-Mohring 2015] generalizes MLTT to include a schema for arbitrary inductive types and their elimination principles. For instance, the natural numbers can be defined in CIC and one can use the natural induction principle to reason about them. CIC, like the Calculus of Constructions [Coquand and Huet 1988], features an additional sort for *propositions*. Terms of a type of the proposition sort have a special status as computationally-irrelevant information that can be erased through extraction [Letouzey 2004]. Additionally, the sort of propositions is impredicative, in contrast to the sort of types for computationally-relevant

terms, and therefore the sort is not structured as a hierarchy. In Coq, these two sorts are called Type and Prop, respectively.

2.1 Syntax and Typing

MuTT and parametrization. Multiverse Type Theory (MuTT) is an extensible variant of MLTT with multiple sorts. At its core, MuTT features dependent functions and universes, together with an extensible framework to define multiple sorts, and their inhabitants. This means that the formal presentation of MuTT is deeply parametrized by a pair $\mathcal{P} = (\mathbb{S}, \Sigma)$:

- MuTT is parametrized by a set $\mathbb{S}$ of sorts, with a distinguished sort $\mathbb{ty}$ (read "type"). The sort $\mathbb{ty}$ is primordial: it is necessarily present, and serves as the recipient to all universes, whatever their sort. A parametrization of MuTT can include additional sorts. To any sort $s \in \mathbb{S}$ corresponds a hierarchy $\square_i^s$ of universes, where s is the sort of the universe and i its level ($i \in \mathbb{N}$). Hierarchies in MuTT are always predicative. Any universe $\square_i^s$ has sort $\mathbb{ty}$ at level $i+1$, or equivalently, has type $\square_{i+1}^{\mathbb{ty}}$. Additionally, the set $\mathbb{S} \setminus \mathbb{ty}$ comes with a predicate isolated(s) which characterizes sorts whose information cannot be used in $\mathbb{ty}$.
- To populate the sorts in $\mathbb{S}$, MuTT is parametrized by two sets underlying the signature Σ: a set of constants C and a set of rewrite rules $\mathcal{R}$, which specify the computational aspect of the constants. All judgments of MuTT are relative to the well-formed signature Σ that guarantees the well-formedness of types, constructors and eliminators, as well as determinism of the reductions and their completeness when the sorts involved are not isolated.

Syntax and notations. The syntax of MuTT, which is mostly standard except for the sort annotation on binders and universes, and the constants c and d from C (explained later on):

$$\begin{array}{lllll} \text{Terms} & t, u, p, A, B & ::= & x \mid \lambda(x :^s A).t \mid t\ u \mid c(\overline{t}) \mid d(\overline{t}; u) \mid \Pi\,(x :^s A)\,B \mid \square^s \\ \text{Substitutions} & \sigma, \overline{t} & ::= & ! \mid (\sigma, t) \mid t_1, \ldots, t_n \end{array}$$

The empty substitution is noted !, and (σ, t) is the extension of a substitution σ with a term t. We use overlined variables $\overline{x}$ to denote a substitution as a sequence of terms or variables x_i and sometimes abuse context notations $\overline{x} : \Gamma$ to make explicit the name of the variables bound in the context Γ. If Γ is a context, Γ_i is its component at the ith position and $\Gamma_{<i}$ is its prefix excluding Γ_i. As usual, we write $A \to B$ for the non-dependent version of the dependent product. We use the isomorphism between typing contexts and telescopes implicitly, i.e. if $\Gamma \vdash t : \Pi\,\Delta, A$ then we can talk about $\overline{u} : \Delta$ a well-typed instance/substitution for the context/telescope Δ.

Judgments. Figure 1 collects the defining judgments of MuTT, which are all parametrized by a well-formed signature Σ. To account for different sorts, traditional judgments have to be augmented with information about the sort. For instance, the formation rule of dependent product should mention both the sorts and the universe levels (Rule Π-explicit-level):

$$\textsc{Π-explicit-level}\ \frac{\Gamma \vdash A :^{\mathbb{ty}} \square_i^{s_1} \qquad \Gamma, x :^{s_1,i} A \vdash B :^{\mathbb{ty}} \square_j^{s_2}}{\Gamma \vdash \Pi\,(x :^{s_1,i} A).\ B :^{\mathbb{ty}} \square^{s_2}_{\max(i,j)}} \qquad \textsc{Π-implicit-level}\ \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x :^{s_1} A \vdash B : s_2}{\Gamma \vdash \Pi\,(x :^{s_1} A).\ B : s_2}$$

Because the formal treatment of universe levels is an orthogonal concern that would obscure the presentation of the multisorted extension of type theory, we adopt typical ambiguity [Whitehead and Russell 1910] and do not explicitly bind universe levels i in the rest of the paper, unless it helps understand the constructions. Hence, we simply use a presentation as in Rule Π-implicit-level. This presentation makes more salient the main information regarding the sorted version of Π: *the sort of a* Π *is the sort of its codomain.* Therefore the typing judgment is written $\vdash t :^s A$. Likewise,

$\Sigma\,;\Gamma \vdash$	Γ is a well formed context with respect to signature Σ
$\Sigma\,;\Gamma \vdash \sigma : \Delta$	σ is a well formed substitution from Γ to Δ
$\Sigma\,;\Gamma \vdash A : s$	A is a well formed type at sort $s \in \mathbb{S}$ in context Γ
$\Sigma\,;\Gamma \vdash t :^s A$	t is a well formed term of type A in context Γ
$\Sigma\,;\Gamma \vdash A \equiv B : s$	A and B are convertible types at sort s in context Γ
$\Sigma\,;\Gamma \vdash t \equiv u :^s A$	t and u are convertible at type A and sort s in context Γ
$\Sigma\,;\Gamma \vdash_d t :^s A$	t is a well formed pattern of type A in context Γ for destructor d

Fig. 1. Judgments of MuTT

$$\textsc{Ctx-Nil}\quad \Sigma\,;\cdot \vdash \qquad \textsc{Ctx-Cons}\quad \frac{\Sigma\,;\Gamma \vdash A : s}{\Sigma\,;\Gamma, x :^s A \vdash} \qquad \textsc{Empty-Sub}\quad \Sigma\,;\Gamma \vdash ! : \cdot \qquad \textsc{Cons-Sub}\quad \frac{\Sigma\,;\Gamma \vdash \sigma : \Delta \qquad \Sigma\,;\Gamma \vdash t :^s A[\sigma]}{\Sigma\,;\Gamma \vdash (\sigma, t) : (\Delta, x :^s A)}$$

$$\textsc{Var}\quad \frac{\Sigma\,;\Gamma \vdash A : s}{\Sigma\,;\Gamma, x :^s A \vdash x :^s A} \qquad \textsc{Weak}\quad \frac{\Sigma\,;\Gamma \vdash t :^{s_1} A \qquad \Sigma\,;\Gamma \vdash B : s_2}{\Sigma\,;\Gamma, y :^{s_2} B \vdash t :^{s_1} A} \qquad \textsc{Conv}\quad \frac{\Sigma\,;\Gamma \vdash t :^s A \qquad \Sigma\,;\Gamma \vdash A \equiv B : s}{\Sigma\,;\Gamma \vdash t :^s B}$$

$$\Pi\text{-}\textsc{Wf}\quad \frac{\Sigma\,;\Gamma \vdash A : s_1 \qquad \Sigma\,;\Gamma, x :^{s_1} A \vdash B : s_2}{\Sigma\,;\Gamma \vdash \Pi\,(x :^{s_1} A).\ B : s_2}$$

$$\Pi\text{-}\textsc{Intro}\quad \frac{\Sigma\,;\Gamma, x :^{s_1} A \vdash t :^{s_2} B}{\Sigma\,;\Gamma \vdash \lambda(x :^{s_1} A).\ t :^{s_2} \Pi\,(x :^{s_1} A).\ B} \qquad \Pi\text{-}\textsc{Elim}\quad \frac{\Sigma\,;\Gamma \vdash t :^{s_2} \Pi\,(x :^{s_1} A).\ B \qquad \Sigma\,;\Gamma \vdash u :^{s_1} A}{\Sigma\,;\Gamma \vdash t\ u :^{s_2} B[u/x]}$$

$$\textsc{Univ-Wf}\quad \frac{s \in \mathbb{S} \qquad \Sigma\,;\Gamma \vdash}{\Sigma\,;\Gamma \vdash \square^s : \mathbb{ty}} \qquad \textsc{Univ-El}\quad \frac{\Sigma\,;\Gamma \vdash A :^{\mathbb{ty}} \square^s}{\Sigma\,;\Gamma \vdash A : s} \qquad \textsc{El-Univ}\quad \frac{\Sigma\,;\Gamma \vdash A : s}{\Sigma\,;\Gamma \vdash A :^{\mathbb{ty}} \square^s}$$

$$\textsc{Inert-Type}\quad \frac{K \in \Sigma \qquad \mathrm{cod}(K) = \square^s \qquad \Sigma\,;\Gamma \vdash \overline{t} : \mathrm{params}(K)}{\Sigma\,;\Gamma \vdash K(\overline{t}) : s}$$

$$\textsc{Inert-Term}\quad \frac{c \in \Sigma \qquad \mathrm{cod}(c) = K(\overline{u}) \qquad \Sigma\,;\Gamma \vdash \overline{p} : \mathrm{params}(c) \qquad \Sigma\,;\Gamma \vdash \overline{t} : \mathrm{dom}(c)[\overline{p}]}{\Sigma\,;\Gamma \vdash c(\overline{p}, \overline{t}) :^s K(\overline{u})[\overline{p}]}$$

$$\textsc{Active-Term}\quad \frac{d \in \Sigma \qquad \Sigma\,;\Gamma \vdash \overline{t} : \mathrm{params}(d) \qquad \Sigma\,;\Gamma \vdash u :^{\mathbb{S}^{\mathrm{dom}}(d)} \mathrm{dom}(d)[\overline{t}]}{\Sigma\,;\Gamma \vdash d(\overline{t}; u) :^{\mathbb{S}^{\mathrm{cod}}(d)} \mathrm{cod}(d)[\overline{t}, u]}$$

Fig. 2. MuTT typing rules (universe levels omitted)

the usual judgment $\vdash A$ of MLTT which says that A is a type is annotated with its sort $\vdash A : s$. Other judgments are decorated similarly.

Typing. Figure 2 adapts the standard rules of MLTT to account for multiple sorts and universe hierarchies. The rules for well-formed type environment, substitution, variables, dependent product, conversion, and universe are all standard from MLTT, but extended with the sort information. For

instance, Rule Π-Intro specifies that a lambda term that takes an argument of type A in sort s_1 can be typed in another sort s_2 provided its body is. Likewise, an application can happen in any sort s_2, even if the argument is typed in sort s_1. Rule Univ-Wf states that all universes are of sort $\mathbb{ty}$. The other uniform choice would assign the sort s to $\square^s$, but this rule is not valid in general, for instance the sort $\mathbb{P}$ of proof-irrelevant types of Coq has actually sort $\mathbb{ty}$ as proof-irrelevant types themselves are not proof-irrelevant and cannot be assigned sort $\mathbb{P}$ itself. Following Coquand's reformulation of Russell's style presentation of universes [Coquand 2019; Sterling 2019], Rules Univ-El and El-Univ together state that a type at sort s can equivalently be seen as a term of type $\square^s$.

The last three rules of Figure 2 deal with the parametrization of MuTT with a set of constants $\mathcal{C}$, as mentioned above. Constants are further classified depending on whether they are *inert* or *active*, *i.e.,* $\mathcal{C} = \mathcal{I} \cup \mathcal{A}$. An inert constant $c \in \mathcal{I}$ does not trigger computation (*e.g.,* a type, such as `List`, or a constructor, such as `cons`), while an active constant $d \in \mathcal{A}$ must come with suitable rewrite rules that specify its computational content (*e.g.,* the elimination principle of an inductive type, such as `listRec`). We now examine each in turn, using lists as a concrete parametrization example.

Inert constants. Inert constants are used to introduce new types (Rule Inert-Type), commonly noted K, as well as new constructors for types (Rule Inert-Term). An inert constant c is described by contexts $\mathsf{params}(c)$ and $\mathsf{dom}(c)$ as well as a type $\mathsf{cod}(c)$ specifying the parameters, the domain of recursive occurences and codomain of the inert constant. The distinction between inert types and constructors is done relatively to the codomain $\mathsf{cod}(c)$, for which there are only two possibilities: when $\mathsf{cod}(c) = \square^s$, then c is an inert type of sort $s \in \mathbb{S}$, and when $\mathsf{cod}(c) = K(\overline{u})$, then c is a constructor of an inert type K . An applied inert type $K(\overline{t})$ is well-typed when K appears in the signature Σ and its arguments are of type $\mathsf{params}(K)$. Note that by the condition of well-formedness of Σ, we know that $\mathsf{dom}(K) = \cdot$. An applied inert term $c(\overline{p}, \overline{t})$ of an inert type K has type $K(\overline{u})[\overline{p}]$ when c appears in Σ, its arguments $\overline{p}$ are of type $\mathsf{params}(c)$, and the recursive occurrences $\overline{t}$ are of type $\mathsf{dom}(c)$. Note that here the substitution $\overline{u}$ providing the arguments of K is used to mediate between $\mathsf{params}(c)$ and $\mathsf{params}(K)$, which may be different, in particular when encoding inductive types with indices.

Example 1. Consider the presentation of lists as an inert type of sort $\mathbb{ty}$. The type `List` is simply given by $\mathsf{cod}(\texttt{List}) = \square^{\mathbb{ty}}$ and $\mathsf{params}(\texttt{List}) = A :^{\mathbb{ty}} \square^{\mathbb{ty}}$ making any $\texttt{List}\,A$ with $A : \mathbb{ty}$ well typed. The constructor `cons` is given by $\mathsf{cod}(\texttt{cons}) = \texttt{List}\,A$, $\mathsf{params}(\texttt{cons}) = A :^{\mathbb{ty}} \square^{\mathbb{ty}}, a :^{\mathbb{ty}} A$ and $\mathsf{dom}(\texttt{cons}) = l :^{\mathbb{ty}} \texttt{List}\,A$. In that case, the substitution $\overline{u}$ from $\mathsf{params}(\texttt{cons})$ to $\mathsf{params}(\texttt{List})$ is the first projection. Using Rule Inert-Term, we get that $\texttt{cons}(A, a, l)$ is well-typed as an inhabitant of $\texttt{List}\,A$ as soon as A is a type in $\mathbb{ty}$, a is an inhabitant of A and l is a list of A itself. Similarly for `nil`, we define $\mathsf{cod}(\texttt{nil}) = \texttt{List}\,A$ and $\mathsf{params}(\texttt{nil}) = A :^{\mathbb{ty}} \square^{\mathbb{ty}}$, leaving $\mathsf{dom}(\texttt{nil})$ empty.

Active constants. An active constant is described by four parameters $\mathsf{params}(d)$, $\mathsf{dom}(d)$, $\mathsf{cod}(d)$ and $\mathbb{S}^{\mathsf{cod}}(d)$. $\mathsf{params}(d)$ corresponds to the (possibly empty) context of parameters of d. The domain $\mathsf{dom}(d)$ specifies the type of the scrutinee of d. As for inert terms, we restrict the system to accept only two alternatives for $\mathsf{dom}(d)$: either $\mathsf{dom}(d) = \square^s$, in which case we define $\mathbb{S}^{\mathsf{dom}}(d) = \mathbb{ty}$; or $\mathsf{dom}(d) = K(\overline{t'})$ with $K \in \Sigma$ and $\mathsf{cod}(K) = \square^s$, in which case we define $\mathbb{S}^{\mathsf{dom}}(d) = s$. An active term $d(\overline{t}; u)$ is well typed (Rule Active-Term) when d appears in Σ, $\overline{t}$ is of type $\mathsf{params}(d)$, the scrutinee u is of type $\mathsf{dom}(d)[\overline{t}]$. When this is the case, its return type is $\mathsf{cod}(d)[\overline{t}, u]$.

Example 2. Coming back to the list example, the eliminator `listRec` is presented as a term destructor with $\mathsf{dom}(\texttt{listRec}) = \texttt{List}\,A$, $\mathsf{params}(\texttt{listRec}) = A :^{\mathbb{ty}} \square^{\mathbb{ty}}, P :^{\mathbb{ty}} A \to \square^{\mathbb{ty}}, p_{\texttt{nil}} :^{\mathbb{ty}} P\ (\texttt{nil}\ A), p_{\texttt{cons}} :^{\mathbb{ty}} \Pi\,(a :^{\mathbb{ty}} A, l :^{\mathbb{ty}} \texttt{List}\,A).\ P\ l \to P\ (\texttt{cons}(A, a, l))$ and with $\mathbb{S}^{\mathsf{cod}}(\texttt{listRec}) = \mathbb{ty}$ and $\mathsf{cod}(\texttt{listRec}) = P\ u$, where u is the variable associated to the scrutinee $u :^{\mathbb{ty}} \texttt{List}\,A$ in Rule Active-Term.

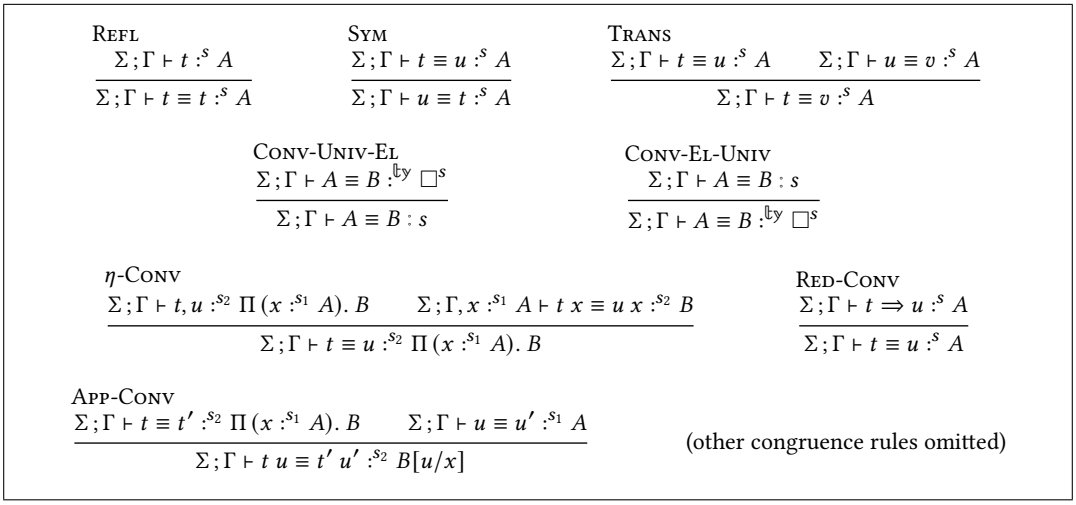

Fig. 3. Conversion for MuTT

2.2 Conversion, Rewrite Rules, Reduction

Conversion. Fig. 3 presents the conversion rules of MuTT. To insist on the central role of reduction in the theory, conversion is defined as the transitive, reflexive, symmetric closure by congruence of reduction. Rule App-Conv illustrates the congruence rule for application. Reduction is embedded inside conversion through Rule Red-Conv. Conversion additionally provides a way to navigate between type and term conversion (Rules Conv-Univ-El and Conv-El-Univ) and also features η-conversion for functions (Rule η-Conv). The definition of reduction itself is parametrized by *rewrite rules* [Cockx et al. 2021] that can be added to MuTT.

Reduction. Fig. 4 describes the notion of reduction in MuTT, which features the usual notion of β-reduction (Rule β-Red). Because reduction is itself typed, one needs to add a compatibility rule with conversion at the level of types (Rule Conv-Red). Congruence rules on applications and active terms are turned into substitution rules in order to emulate reduction to *weak-head normal forms* (whnfs). Indeed, the classification between inert $c \in \mathcal{I}$ and active constants $d \in \mathcal{A}$ gives rise to well-behaved definitions of whnfs and *neutral forms*, two key notions to establish the metatheory.

$$\begin{array}{lllll} \text{whnf } t & w & ::= & ne \mid \Pi\,(x :^s A)\,B \mid \lambda(x :^s A).\,t \mid \square^s \mid c(\overline{p}, \overline{t}) & \\ \text{neutral } t & ne & ::= & x \mid ne\ t \mid d(\overline{t}; ne) \mid d(\overline{t}; c(\overline{p}, \overline{u})) & \neg\text{react}(d, c, \mathcal{R}) \end{array}$$

These notions are at the heart of the logical relation given in §5. For now, it is enough to know that weak-head normal forms correspond to terms that can *not* be head-reduced, of which neutral terms are the particular cases where the term may not stay in weak-head normal form after substitution. An inert type or inert term is always a whnf. An active term is neutral when its scrutinee is in whnf and there is no rewrite rule in $\mathcal{R}$ that can be fired, as expressed by the following definition:

$$\text{react}(d, c, \mathcal{R}) \quad \stackrel{\text{def}}{=} \quad \exists(\sigma : \Delta \hookrightarrow \Delta_{\text{lin}}, d(\overline{x}; pat), r) \in \mathcal{R}, \quad pat = c\ q$$

So to achieve whnf reduction, we need to add a substitution rule (Rule App-Subs) that reduces the left-hand side of an application and the scrutinee of an active term until it reaches a whnf. Finally, each rewrite rule in $\mathcal{R}$ is turned into a reduction rule using rule Rew-Red, which basically considers any correct (linear) instantiation of a rewrite rule.

$$\frac{\Sigma\,;\Gamma, x :^{s_1} A \vdash t :^{s_2} B \qquad \Sigma\,;\Gamma \vdash u :^{s_1} A}{\Sigma\,;\Gamma \vdash (\lambda(x :^{s_1} A).\, t)\; u \Rightarrow t[u/x] :^{s_2} B[u/x]}\;\beta\text{-Red}$$

$$\frac{\Sigma\,;\Gamma \vdash t \Rightarrow u :^{s} A \qquad \Sigma\,;\Gamma \vdash A \equiv B : s}{\Sigma\,;\Gamma \vdash t \Rightarrow u :^{s} B}\;\textsc{Conv-Red}$$

$$\frac{\Sigma\,;\Gamma \vdash t \Rightarrow t' :^{s_2} \Pi\,(x :^{s_1} A).\, B \qquad \Sigma\,;\Gamma \vdash u :^{s_1} A}{\Sigma\,;\Gamma \vdash t\; u \Rightarrow t'\; u :^{s_2} B[u/x]}\;\textsc{App-Subs}$$

$$\frac{\Sigma\,;\Gamma \vdash \overline{p} : \mathrm{params}(d) \qquad \Sigma\,;\Gamma \vdash t \Rightarrow t' :^{\mathbb{S}^{\mathrm{dom}}(d)} \mathrm{dom}(d)[\overline{p}]}{\Sigma\,;\Gamma \vdash d(\overline{p}; t) \Rightarrow d(\overline{p}; t') :^{\mathbb{S}^{\mathrm{cod}}(d)} \mathrm{cod}(d)[\overline{p}, a]}\;\textsc{Active-Subs}$$

$$\frac{(\sigma : \Delta \hookrightarrow \Delta_{\mathrm{lin}}, d(\overline{x}; pat), r) \in \mathcal{R} \qquad \Sigma\,;\Gamma \vdash \sigma' : \Delta_{\mathrm{lin}} \qquad (\overline{t}, a) = (\overline{x}, \epsilon(pat))[\sigma'] \qquad \Sigma\,;\Gamma \vdash d(\overline{t}, a) :^{\mathbb{S}^{\mathrm{cod}}(d)} \mathrm{cod}(d)[\overline{t}, a]}{\Sigma\,;\Gamma \vdash d(\overline{t}; a) \Rightarrow r[\sigma', {\rho^{\mathsf{rec}}}_d(pat)[\sigma']] :^{\mathbb{S}^{\mathrm{cod}}(d)} \mathrm{cod}(d)[\overline{t}, a]}\;\textsc{Rew-Red}$$

Fig. 4. Reduction Rules for MuTT

Patterns and Rewrite Rules. Active constants are interesting when associated to rewrite rules, which make it possible to extend the conversion of MuTT. A rewrite rule

$$(\sigma : \Delta \hookrightarrow \Delta_{\mathrm{lin}}, d(\overline{x}; pat), r) \in \mathcal{R}$$

is given by a left-hand side, characterized by its head symbol d, which must be an active constant, a renaming $\overline{x}$, and a pattern pat for its scrutinee, and a right-hand side r, which is simply a term. It ensures that its left- and right-hand sides are convertible for any instance of the (linear) context Δ_{lin} for which the left-hand side is well-typed (Rew-Red in Fig. 4). However, in general, the use of a linear context is not enough to guarantee that the right-hand side is well typed, because this may rely on some auxiliary conversions ensured by typing. On the other hand, the use of a non-linear context alone is not sufficient either because the reduction rule Rew-Red would require a prohibitive use of conversion in the definition of reduction. To remedy to this tension, a rewrite rule has two contexts Δ and Δ_{lin} with a renaming σ (*i.e.,* a substitution with only variables). The point of the context Δ and substitution σ is to provide a non-linear version of the rewrite rule, which can be used for typing purposes, with a side condition that every well-typed linear occurrence of the rule is actually well-typed as a non-linear occurrence. When the rewrite rule is *linear*, that is when $\Delta = \Delta_{\mathrm{lin}}$ and $\sigma = id$, we simply write $(\Delta, d(\overline{x}; pat), r)$.

The syntax and typing rules of patterns are described in Fig. 5. A pattern consists either of an inert constant $c \in C$ or a Π applied to metavariables, while a metavariable can be either a variable x or a recursive occurrence $?^{\mathsf{rec}} z[\sigma]$. The purpose of recursive occurrences is twofold. When typing a pattern, a recursive occurrence has the type of the domain head symbol $\mathrm{dom}(d)$, when its substitution σ is a correct substitution for the parameters of d (Rule MetaVarRec-Pat-Term). It additionally enforces that the recursive occurrences $?^{\mathsf{rec}} z[\sigma]$ correspond to a variable z with the same type in the context, which may be used in the right-hand side of a rewrite rule. Apart from these two rules, the typing rules of patterns just mimic the typing rules for terms and substitutions.

From the pattern pat, we define three functions: a telescope $\mathtt{rec}_d(pat)$ extending the context Δ, collecting the recursive occurrences in the pattern, which is used to typecheck the recursive calls to d in the right-hand side of the rewrite rule; a substitution $\Sigma\,;\Delta \vdash {\rho^{\mathsf{rec}}}_d(pat) : \mathtt{rec}_d(pat)$

$$p \;::=\; c(q_1, \ldots, q_n) \mid \Pi^s\ q_1\ q_2 \qquad \text{(pattern)}$$
$$q \;::=\; x \mid ?^{\mathsf{rec}} z[\sigma] \qquad \text{(meta-variable)}$$

$\boxed{\Sigma ; \Gamma \vdash_d\ p :^s A}$ *Pattern typing.*

$$\textsc{Inert-Pat}\ \frac{\Sigma ; \Gamma \vdash_d\ \overline{q} : \mathsf{params}(c) \qquad \Sigma ; \Gamma \vdash_d\ \overline{q'} : \mathsf{dom}(c)[\overline{\epsilon(q)}]}{\Sigma ; \Gamma \vdash_d\ c(\overline{q}, \overline{q'}) :^{\mathbb{S}^{\mathsf{cod}}(c)}\ \mathsf{cod}(c)[\overline{\epsilon(q)}]} \qquad \textsc{MetaVar-Pat}\ \frac{x :^s A \in \Gamma}{\Sigma ; \Gamma \vdash_d\ x :^s A}$$

$$\textsc{Sub-Pat}\ \frac{\Sigma ; \Gamma \vdash_d\ \sigma : \Delta \qquad \Sigma ; \Gamma \vdash_d\ t :^s A[\sigma]}{\Sigma ; \Gamma \vdash_d\ (\sigma, t) : (\Delta, x :^s A)} \qquad \Pi^{s_1}\textsc{-Pat}\ \frac{\Sigma ; \Gamma \vdash_d\ q_1 :^{\mathbb{ty}} \square^{s_1} \qquad \Sigma ; \Gamma \vdash_d\ q_2 :^{\mathbb{ty}} \epsilon(q_1) \to \square^{s_2}}{\Sigma ; \Gamma \vdash_d\ \Pi^{s_1}\ q_1\ q_2 :^{\mathbb{ty}} \square^{s_2}}$$

$$\textsc{MetaVarRec-Pat-Term}\ \frac{\Sigma ; \Gamma \vdash \sigma : \mathsf{params}(d) \qquad z :^s \mathsf{dom}(d)[\sigma] \in \Gamma}{\Sigma ; \Gamma \vdash_d ?^{\mathsf{rec}} z[\sigma] :^s \mathsf{dom}(d)[\sigma]}$$

$\boxed{\epsilon(pat), \mathtt{rec}_d(pat), {\rho^{\mathsf{rec}}}_d(pat)}$ *erasure to terms, context and substitution of recursive hypothesis.*

$$\begin{array}{llll}
\epsilon(x) & := x & \mathtt{rec}_d(x) & := \cdot \\
\epsilon(?^{\mathsf{rec}} z[\sigma]) & := z & \mathtt{rec}_d(?^{\mathsf{rec}} z[\sigma]) & := z^{\mathsf{rec}} : \mathsf{cod}(d)[\sigma, z] \\
\epsilon(\Pi^s\ q_1\ x) & := \Pi\,(y :^s \epsilon(q_1))\ (x\ y) & \mathtt{rec}_d(\Pi^s\ q_1\ x) & := \mathtt{rec}_d(q_1) \\
\epsilon(\Pi^s\ q_1\ ?^{\mathsf{rec}} z[\sigma]) & := \Pi\,(y :^s \epsilon(q_1))\ (z\ y) & \mathtt{rec}_d(\Pi^s\ q_1\ ?^{\mathsf{rec}} z[\sigma]) & := z^{\mathsf{rec}} : \Pi\,(y :^s \epsilon(q_1))\ \mathsf{cod}(d)[\sigma, z\ y] \\
\epsilon(c(q_1, \ldots, q_n)) & := c(\epsilon(q_1), \ldots, \epsilon(q_n)) & \mathtt{rec}_d(c(q_1, \ldots, q_n)) & := \mathtt{rec}_d(q_1), \ldots, \mathtt{rec}_d(q_n)
\end{array}$$

$$\begin{array}{lll}
{\rho^{\mathsf{rec}}}_d(x) & := & ! \\
{\rho^{\mathsf{rec}}}_d(?^{\mathsf{rec}} z[\sigma]) & := & d(\sigma; z) \\
{\rho^{\mathsf{rec}}}_d(\Pi^s\ q_1\ x) & := & {\rho^{\mathsf{rec}}}_d(q_1) \\
{\rho^{\mathsf{rec}}}_d(\Pi^s\ q_1\ ?^{\mathsf{rec}} z[\sigma]) & := & ({\rho^{\mathsf{rec}}}_d(q_1), \lambda\,(y : \epsilon(q_1)).\, d(\sigma, z\ y)) \\
{\rho^{\mathsf{rec}}}_d(c(q_1, \ldots, q_n)) & := & {\rho^{\mathsf{rec}}}_d(q_1), \ldots, {\rho^{\mathsf{rec}}}_d(q_n)
\end{array}$$

Fig. 5. Syntax, typing rules and functions on patterns

$$\textsc{Valid-Rew}\ \frac{\begin{array}{c}\Sigma ; \Delta \vdash \sigma : \Delta_{\mathsf{lin}} \qquad \Sigma ; \Delta_{\mathsf{lin}} \vdash \overline{x} : \mathsf{params}(d) \qquad \Sigma ; \Delta_{\mathsf{lin}} \vdash_d\ pat :^{\mathbb{S}^{\mathsf{dom}}(d)} A \qquad A[\sigma] = \mathsf{dom}(d)[\overline{x}][\sigma] \\ \Sigma ; \Delta, \mathtt{rec}_d(pat)[\sigma] \vdash r :^{\mathbb{S}^{\mathsf{cod}}(d)} \mathsf{cod}(d)[\overline{x}, \epsilon(pat)][\sigma] \qquad \mathsf{linearizable}(\sigma : \Delta \hookrightarrow \Delta_{\mathsf{lin}}, d(\overline{x}; pat), r)\end{array}}{\Sigma ; \sigma : \Delta \hookrightarrow \Delta_{\mathsf{lin}} \vdash d(\overline{x}; pat) \rightsquigarrow r}$$

Fig. 6. Typing for rewrite rules

instantiating all occurrences of $?^{\mathsf{rec}} z[\sigma]$ with the corresponding intended instance of d; and an erasure function $\epsilon(pat)$ computing the underlying term of a pattern.

Finally, Rule Valid-Rew specifies when a rewrite rule is valid (Fig. 6). The rule checks that: (i) the renaming σ is well-typed, (ii) the rewrite rule is linearizable (iii) the parameters $\overline{x}$ form a well-typed renaming to $\mathsf{params}(d)$ in context Δ_{lin}, (iv) the pattern pat is a well-typed pattern in context Δ_{lin}, (v) and that the right-hand side r is a well-typed term, in the context Δ extended with the information that the recursive occurrences appearing in pat are now seen as variables living in the codomain of d (modulo the renaming σ).

In the rule, the context Δ_{lin} is used to typecheck separately the arguments $\overline{x}$ and pat of the left-hand side linearly, ensuring that weak-head reduction is enough to detect when a rewrite rule can fire. However, we need to allow non-linearity (described by the renaming σ) in order to enforce that the types in the left-hand side agree up to the renaming and to type-check the right hand-side, this even for the simple example of lists (see Example 3). To guarantee a posteriori that linear and non-linear matching are equivalent, we introduce the notion of a linearizable rewrite rule.

Definition 1 (Linearizable rewrite rule). *A rewrite rule* $(\sigma : \Delta \hookrightarrow \Delta_{\text{lin}}, d(\overline{x}; pat), r)$ *is linearizable, noted* $\mathsf{linearizable}(\sigma : \Delta \hookrightarrow \Delta_{\text{lin}}, d(\overline{x}; pat), r)$*, when*

(a) every variable in Δ_{lin} *occurs exactly once in* $(\overline{x}, \epsilon(pat))$ *and either*
(b) $\mathsf{dom}(d) = \square^s$ *and the rewrite rule is linear; or*
(b') $\mathsf{dom}(d) = K(\overline{u_d})$ *and the following holds where* $pat = c(\overline{q})$ *and* $\mathsf{cod}(c) = K(\overline{u_c})$*:*

$$\Sigma\,;\Delta_{\text{lin}} \vdash \overline{u_c}[\overline{q}] \equiv \overline{u_d}[\overline{x}] : \mathsf{params}(K) \Rightarrow \exists \tau,\ \tau[\sigma] = \mathrm{id}_\Delta \wedge \Sigma\,;\Delta_{\text{lin}} \vdash \sigma[\tau] \equiv \mathrm{id}_{\Delta_{\text{lin}}} : \Delta_{\text{lin}}.$$

The condition (b) says that every eliminator on a universe must be linear. The condition (b') says that using the conversion constraints collected from the fact that the left-hand side type-checks, one can show that actually the renaming σ admits an inverse, in other word, linear and non-linear matching coincides up to conversion.

The precise type for the recursive occurrences is computed by the function $\mathtt{rec}_d(pat)$. Thus, when typing a rewrite rule, a recursive occurrence $?^{\mathsf{rec}}z[\sigma]$ of a pattern is seen as variable z of type $\mathsf{dom}(d)[\sigma]$ when typing both sides, and additionally as a variable z^{rec} of type $\mathsf{cod}(d)[\sigma, z]$, representing the result of applying d to the variable, when typing the right-hand side. This allows us to encode recursive calls to d that may occur on the right-hand side of a rewrite rule, as illustrated by the following example.

Example 3. Coming back to the representation of lists, the two rewrite rules for `listRec` are:

$$\mathcal{R}_{\mathtt{List}} \stackrel{\text{def}}{=} \left| \begin{array}{l} .\,;\overline{p}, A : \Delta \hookrightarrow (\Delta, B : \square^{\mathbb{ty}}) \vdash \mathtt{listRec}(\overline{p}; \mathtt{nil}\ B) \rightsquigarrow p_{\mathtt{nil}} \\ .\,;\overline{p}, A, a, l : \Delta_{\text{cons}} \hookrightarrow \Delta_{\text{lin}} \vdash \mathtt{listRec}(\overline{p}; \mathtt{cons}(B, a, (?^{\mathsf{rec}} l[\overline{p}]))) \rightsquigarrow p_{\mathtt{cons}}\ a\ l\ l^{\mathsf{rec}} \end{array} \right.$$

with $\overline{p} = A, P, p_{\mathtt{nil}}, p_{\mathtt{cons}}$ and

$$\Delta = \overline{p} : \mathsf{params}(\mathtt{listRec}) \quad \Delta_{\text{cons}} = \Delta, a :^{\mathbb{ty}} A, l :^{\mathbb{ty}} \mathtt{List}\ A \quad \Delta_{\text{lin}} = \Delta, B : \square^{\mathbb{ty}}, a :^{\mathbb{ty}} B, l :^{\mathbb{ty}} \mathtt{List}\ B.$$

The first rewrite rule is valid because as the right-hand side $p_{\mathtt{nil}}$ has type $P\ (\mathtt{nil}\ A)$, and the linearizability condition amounts to show that $\overline{p}, A$ has a retraction, knowing that $A \equiv B$, which is direct by mapping B to A. The second rule makes use of a recursive occurrence $?^{\mathsf{rec}} l[\overline{p}]$, which corresponds both to the variable l of type $\mathtt{List}\ A$ and, for the right-hand side, to the variable l^{rec} of type $P\ l$ representing the recursive call on l. Therefore, the right-hand side is well-typed, with type $P\ (\mathtt{cons}(A, a, l))$. Linearizability of the rule is similar to the case of `nil` .

2.3 Well-formed Signature

We now turn to the definition of a well-formed signature. The signature imposes constraints on each inert and active constant that structure their global behavior and interaction with the whole system. Inert constants K building types are then classified as **positive** if they come with a set $\mathcal{I}_K$ of inert constant called **constructors** to introduce them, or **negative** if they come with a set $\mathcal{A}_K$ of active constants called **observations**. An active constant defined on a positive type is then called an **eliminator**, whereas an inert constant inhabiting a negative type is called a **builder**. Universes are treated as positive types, with the exception that their constructors are open-ended and consist of any type constant of the adequate sort. A well-formed signature is either the empty signature $\cdot$, a well-formed signature Σ extended with a well-formed positive type $(K, \mathcal{I}_K)$, negative type $(K, \mathcal{A}_K)$, an eliminator $(d, \mathcal{R}_d)$ on a universe or positive type, or a builder $(c, \mathcal{R}_c)$ on a negative type. We say

that a constant or a rewrite rule κ belongs to a well-formed signature Σ, noted $\kappa \in \Sigma$, if it appears in one of its components. Well-formedness for rewriting relies on three key properties—determinism of the set of rewrite rules, progress and isolation—detailed next.

Deterministic rewrite rules. The first property ensures that the notion of whnf reduction defined in Fig. 4 is deterministic, which is crucial to easily get confluence of the system.[1]

Definition 2 (Deterministic rewrite rules). *A set of rewrite rules $\mathcal{R}$ is deterministic, noted* $\mathrm{det}(\mathcal{R})$*, if any two rewrite rules from $\mathcal{R}$ with common head symbol and common head-constant in their patterns have the same right-hand sides:*

$$\mathrm{det}(\mathcal{R}) \stackrel{\text{def}}{=} \forall(\sigma : \Delta \hookrightarrow \Delta_{\text{lin}}, d(\overline{x}; pat), r), (\sigma' : \Delta' \hookrightarrow \Delta'_{\text{lin}}, d'(\overline{x'}; pat'), r') \in \mathcal{R},$$
$$d = d' \ \wedge \ pat = c\ q \ \wedge \ pat' = c\ q' \ \Rightarrow \ r = r'$$

Rewrite progress. The second notion that helps us characterize well-formed signatures is *progress* between a set of inert terms and a set of active terms with respect to a set of rewrite rules.

Definition 3 (Rewrite progress). *A set of inert constants I and a set of active constants A satisfy rewrite progress with respect to a set of rewrite rules $\mathcal{R}$, noted* $\mathrm{progress}(I, A, \mathcal{R})$*, if every active constants in A reacts to every constant in I according to $\mathcal{R}$:*

$$\mathrm{progress}(I, A, \mathcal{R}) \stackrel{\text{def}}{=} \forall c \in I, d \in A, \ \mathrm{react}(d, c, \mathcal{R})$$

When $I = \{c\}$ or $A = \{d\}$ are singleton, we note respectively $\mathrm{progress}(c, A, \mathcal{R})$ *and* $\mathrm{progress}(I, d, \mathcal{R})$.

Isolated sorts. When there exists an active term d for which rewrite progress does not hold (with respect to its associated inert constants and rewrite rules), the notion of canonicity, as defined in Theorem 5, is in danger. To guarantee that in this case, canonicity in $\mathbb{ty}$ is still valid, we rely on the notion of *isolated sort*, which ensures that there is no "leak" from the sort hosting d into $\mathbb{ty}$. In counterpart, well-formedness condition on eliminators from isolated sorts needs to ensure that isolation is preserved. In a well-formed signature Σ the following invariant will be maintained:

$$d \in \Sigma \ \wedge \ \mathrm{isolated}(s) \quad \Longrightarrow \quad \mathrm{isolated}(s')$$

where d is an active constant and $\mathrm{dom}(d) = \square^s \vee \mathbb{S}^{\mathrm{dom}}(d) = s$ and $\mathrm{cod}(d) = \square^{s'} \vee \mathbb{S}^{\mathrm{cod}}(d) = s'$.

We can now turn to the definition of well-formed positive types, negative types, eliminators (of positive types) and builders (of negative types).

Definition 4 (Well-formed positive type). *An inert constant K building a type of sort s, $\mathrm{cod}(K) = \square^s$, together with its constructors $\mathcal{I}_K$ is a well formed positive type in the signature Σ when*

(1) There is no active constant $d \in \Sigma$ defined on $\square^s$

$$\forall d \in \mathcal{A}, \quad d \in \Sigma \ \Rightarrow \ \mathrm{dom}(d) \neq \square^s$$

(2) Its parameters are well-formed $\Sigma\,; \mathrm{params}(K) \vdash$ and domain is empty $\mathrm{dom}(K) = \cdot$

(3) Any inert *constant $c \in \mathcal{I}_K$ building a term in K, $\mathrm{cod}(c) = K(\overline{u})$, has parameters, domain and codomain well-formed in Σ:*

$$\Sigma\,; \mathrm{params}(c) \vdash \qquad \forall i, \mathrm{dom}(c)_i = K(\overline{t}) \wedge \Sigma\,; \mathrm{params}(c) \vdash \overline{t} : \mathrm{params}(K)$$

$$\Sigma\,; \mathrm{params}(c) \vdash \overline{u} : \mathrm{params}(K)$$

[1] We could adopt a more permissive condition for confluence [Cockx et al. 2021], but this is not central here.

Rule (1) forces new type constructors of a sort to be checked before eliminators on the universe. Rule (2) says that an inert type has only well-formed parameters. Rule (3) checks that every inert constant populating K have well-formed parameters and strictly positive occurrences of arguments in K.

Definition 5 (Well-formed negative type). *An inert constant K building a type of sort s, $\mathrm{cod}(K) = \square^s$, together with its observations $\mathcal{A}_K$ is a well formed negative type in the signature Σ when*

(1) There is no active constant $d \in \Sigma$ defined on $\square^s$

$$\forall d \in \mathcal{A}_{,} \quad d \in \Sigma \implies \mathrm{dom}(d) \neq \square^s$$

(2) Its parameters are well-formed $\Sigma\,;\mathrm{params}(K) \vdash$ and domain is empty $\mathrm{dom}(K) = \cdot$

*(3) $\mathcal{A}_K$ is an ordered set of **active** constant $\overline{d}$ that share the same parameters as K, $\forall i, \mathrm{params}(d_i) = \mathrm{params}(K)$. Any $d_i \in \mathcal{A}_K$ has domain K, $\mathrm{dom}(d) = K(\mathrm{id}_{\mathrm{params}(K)})$, and well-formed codomain in Σ that can depend on the result of previous $\overline{d}_{<i}$:*

$$\begin{cases} \Sigma\,;\mathrm{params}(K), \overline{\mathrm{cod}(d)}_{<i} \vdash \overline{t} : \mathrm{params}(K) & \text{if } \mathrm{cod}(d_i) = K(\overline{t}) \\ \Sigma\,;\mathrm{params}(K), \overline{\mathrm{cod}(d)}_{<i} \vdash \mathrm{cod}(d_i) : \mathbb{S}^{\mathrm{cod}}(d_i) & \text{otherwise} \end{cases}$$

(4) If s is isolated then any $d \in \mathcal{A}_K$ land in an isolated sort:

$$\forall s' \in \mathbb{S}, d \in \mathcal{A}_d, \mathrm{isolated}(s) \wedge (\mathrm{cod}(d) = \square^{s'} \vee \mathbb{S}^{\mathrm{cod}}(d) = s') \Rightarrow \mathrm{isolated}(s')$$

The two first rules are the same as for positive types, and the last rule dually checks active constants. More specifically, the condition that all active constants shares the same set of parameters $\mathrm{params}(K)$ ensures that those active constants actually define *observations* of inhabitant of K.

Definition 6 (Well-formed eliminator). *An active constant $d \in \mathcal{A}$ together with rewrite rules $\mathcal{R}_d$ is well-formed in signature Σ when the following conditions hold, where*

$$\begin{cases} s_d = s, \quad \mathcal{I}_d = \{\Pi\} \cup \{K \in \Sigma \mid \mathrm{cod}(K) = \square^s\} & \text{if } \mathrm{dom}(d) = \square^s \text{ is a universe} \\ s_d = \mathbb{S}^{\mathrm{dom}}(d), \quad \mathcal{I}_d = \mathcal{I}_K & \text{if } \mathrm{dom}(d) = K(\overline{u}) \text{ is a positive type in } \Sigma \end{cases}$$

(1) If s_d is isolated, d must land in an isolated sort, otherwise it must satisfy progress:

$$\begin{cases} (\mathrm{cod}(d) = \square^{s'} \vee \mathbb{S}^{\mathrm{cod}}(d) = s') \wedge \mathrm{isolated}(s') & \text{if } \mathrm{isolated}(s_d) \\ \mathrm{progress}(d, \mathcal{I}_d, \mathcal{R}_d) & \text{otherwise} \end{cases}$$

(2) Its parameters, domain and codomains are well-formed in Σ

$$\mathrm{progress}(d, \mathcal{I}_d, \mathcal{R}_d) \vee \mathrm{isolated}(s_d) \qquad \Sigma\,;\mathrm{params}(d) \vdash$$

$$\Sigma\,;\mathrm{params}(d) \vdash \mathrm{dom}(d) : \mathbb{S}^{\mathrm{dom}}(d) \qquad \Sigma\,;\mathrm{params}(d), x :^{\mathbb{S}^{\mathrm{dom}}(d)} \mathrm{dom}(d) \vdash \mathrm{cod}(d) : \mathbb{S}^{\mathrm{cod}}(d)$$

(3) The rewrite rules in $\mathcal{R}_d$ are deterministic $\det(\mathcal{R}_d)$, have head symbol d and are well-typed

$$\forall (\sigma : \Delta \hookrightarrow \Delta_{\mathrm{lin}}, d(\overline{x}; pat), r) \in \mathcal{R}_d, \quad \Sigma\,;\sigma : \Delta \hookrightarrow \Delta_{\mathrm{lin}} \vdash d(\overline{x}; pat) \rightsquigarrow r$$

Definition 7 (Well-formed builder). *An inert constant $c \in \mathcal{I}$ building a term of negative type $(K, \mathcal{A}_K) \in \Sigma$, that is $\mathrm{cod}(c) = K(\overline{u})$, together with rewrite rules $\mathcal{R}_c$ is well-formed in Σ when:*

(1) It satisfies progress or belongs to an isolated sort and its parameters, domain and codomains are well-formed in Σ:

$$\mathrm{progress}(\mathcal{A}_d, c, \mathcal{R}_c) \vee \mathrm{isolated}(\mathbb{S}^{\mathrm{cod}}(c)) \qquad \Sigma\,;\mathrm{params}(c) \vdash$$

$$\forall i, \Sigma\,;\mathrm{params}(c), \mathrm{dom}(c)_{<i} \vdash \mathrm{dom}(c)_i : \mathbb{S}^{\mathrm{dom}}(c)_i \qquad \Sigma\,;\mathrm{params}(c) \vdash \overline{u} : \mathrm{params}(K)$$

(2) The rewrite rules in $\mathcal{R}_c$ are deterministic $\det(\mathcal{R}_c)$, have patterns with head-constant c and are well-typed

$$\forall(\sigma : \Delta \hookrightarrow \Delta_{\mathrm{lin}}, d(\overline{x}; c(\overline{q})), r) \in \mathcal{R}_c, \quad \Sigma\,;\sigma : \Delta \hookrightarrow \Delta_{\mathrm{lin}} \vdash d(\overline{x}; c(\overline{q})) \rightsquigarrow r$$

Example 4. The type of lists, as other inductive types, is a positive type. Thus, to check that it can be added to a well-formed signature Σ, one need to check that $(\texttt{List}, \{\texttt{nil}, \texttt{cons}\})$ is a well-formed positive types in Σ, and that $(\texttt{listRec}, \mathcal{R}_{\texttt{List}})$ is a well-formed active constant in Σ, $(\texttt{List}, \{\texttt{nil}, \texttt{cons}\})$. The typing conditions for well-formation of $(\texttt{List}, \{\texttt{nil}, \texttt{cons}\})$ and $(\texttt{listRec}, \mathcal{R}_{\texttt{List}})$ can be simply checked in general, as they are independent of Σ in this specific situation, and have already been discussed in the presentation of this example. The last points to check are progress and determinism. This is fairly straightforward as it amounts to check that there is exactly one rule in $\mathcal{R}_{\texttt{List}}$ for the eliminator `listRec` and the constructors `nil` and `cons`.

2.4 Metatheoretical properties of MuTT

We will prove in § 5 that the well-formedness of a signature is sufficient to prove that MuTT enjoys the following metatheoretical properties for any valid parametrization $\mathcal{P} = (\mathbb{S}, \Sigma)$, making it well-suited as the underlying theory of a proof assistant.

Theorem 5 (Canonicity for the $\mathbb{ty}$ hierarchy).

- *If A is a closed type of sort $\mathbb{ty}$, $\Sigma\,;\vdash A : \mathbb{ty}$, then A is convertible to either a universe $\square^s$, a dependent product $\Pi\,(x :^s X)Y$, or an inert constant $K(\overline{t})$.*
- *If t is a closed term of a positive constant type $K(\overline{a})$ in $\mathbb{ty}$, $\Sigma\,;\vdash t :^{\mathbb{ty}} K(\overline{a})$ with $(K, \mathcal{I}_K) \in \Sigma$, then t is convertible to a constructor $c(\overline{p}, \overline{v})$ with $c \in \mathcal{I}_K$.*

Assuming that Σ provides an empty type in $\mathbb{ty}$, that is a positive type $\bot : \mathbb{ty}$ with no introduction $\mathcal{I}_\bot = \varnothing$, we also obtain the logical consistency of $\mathbb{ty}$ for any parametrization.

Theorem 6 (Logical consistency of the $\mathbb{ty}$ hierarchy). *There is no closed proof term e of the empty type $\Sigma\,;\vdash e :^{\mathbb{ty}} \bot$.*

Theorem 7 (Decidability of conversion and typechecking).

- *If $\Sigma\,;\Gamma \vdash A : s$ and $\Sigma\,;\Gamma \vdash B : s$, it is decidable whether $\Sigma\,;\Gamma \vdash A \equiv B : s$ is derivable.*
- *If $\Sigma\,;\Gamma \vdash A : s$, it is decidable whether $\Sigma\,;\Gamma \vdash t :^s A$.*

3 EXPRESSIVITY AND INSTANCES OF THE MULTIVERSE TYPE THEORY

This section develops several instances of parametrization of MuTT, building upon the formal framework described in §2: inductive types (§3.1), a Coq-style sort of propositions (§3.2), Exceptional Type Theory (§3.3), axioms (§3.4), and dependent elimination (§3.5).

3.1 Inductive and Record Types

Section 2 used lists to illustrate the definition of an inductive type; one can easily infer the definition of natural numbers $\mathbb{N}$ as a non-decorated version of lists. We now show that standard Σ-types and identity types can also be represented faithfully in MuTT.

Σ types. Σ types represent dependent pairs. They are, like Π-types, a negative type constructor, defined by its two observations, the first and second projection.

- The type constructor Σ in $\mathbb{ty}$ is given by: $\mathtt{params}(\Sigma) = A :^{\mathbb{ty}} \square_i^{\mathbb{ty}}, B :^{\mathbb{ty}} A \to \square_j^{\mathbb{ty}}$, $\mathtt{cod}(\Sigma) = \square^{\mathbb{ty}}_{\max(i,j)}$
- The projections are active constants `fst` and `snd` with $\mathtt{params}(\mathtt{fst}) = \mathtt{params}(\mathtt{snd}) = \mathtt{params}(\Sigma)$, $\mathtt{dom}(\mathtt{fst}) = \mathtt{dom}(\mathtt{snd}) = \Sigma\ A\ B$ and $\mathtt{cod}(\mathtt{fst}) = A$. For the second projection, $\mathtt{cod}(\mathtt{snd}) = B\ fst$, where $fst : A$ (see definition 5, item 3). This is an example where later projections depend on former ones.
- The default builder constant `pair` is an inert constant presented by $\mathtt{cod}(\mathtt{pair}) = \Sigma\ A\ B$ and $\mathtt{params}(\mathtt{pair}) = \mathtt{params}(\Sigma)$, $a :^{\mathbb{ty}} A, b :^{\mathbb{ty}} B\ a$
- We set $\Delta = \mathtt{params}(\mathtt{pair})$ and $\Delta_{\mathrm{lin}} = \mathtt{params}(\Sigma), C :^{\mathbb{ty}} \square^{\mathbb{ty}}, D :^{\mathbb{ty}} A \to \square^{\mathbb{ty}}, c :^{\mathbb{ty}} C, d :^{\mathbb{ty}} D\ c$
  Ensuring progress we define the projection rewrite rules (deterministic because no overlap):

$$\Sigma, \mathtt{fst}, \mathtt{snd}; A, B, A, B, a, b : \Delta \hookrightarrow \Delta_{\mathrm{lin}} \vdash \mathtt{fst}(A, B; \mathtt{pair}(C, D, c, d)) \rightsquigarrow c$$
$$\Sigma, \mathtt{fst}, \mathtt{snd}; A, B, A, B, a, b : \Delta \hookrightarrow \Delta_{\mathrm{lin}} \vdash \mathtt{snd}(A, B; \mathtt{pair}(C, D, c, d)) \rightsquigarrow d$$

- For typing purpose of the second rewrite rule: observe that (after the action of the Δ_{lin} to Δ substitution) b has type $B\ a$ according to the typing rule for `pair`. This type is convertible to $B\ (\mathtt{fst}(A, B; \mathtt{pair}(A, B, a, b)))$ thanks to the rewrite rule for `fst`.

Identity types. Illustrating the expressivity of our framework, we can also define standard Martin-Löf identity types `Id` with the `J` elimination rule of Paulin-Mohring [1993].

- $\mathtt{params}(\mathtt{Id}) = A :^{\mathbb{ty}} \square_i^{\mathbb{ty}},\ a :^{\mathbb{ty}} A,\ x :^{\mathbb{ty}} A$ and $\mathtt{cod}(\mathtt{Id}) = \square_i^{\mathbb{ty}}$
- The unique constructor is `refl`, with $\mathtt{params}(\mathtt{refl}) = A :^{\mathbb{ty}} \square_i^{\mathbb{ty}}, a :^{\mathbb{ty}} A$ and $\mathtt{cod}(\mathtt{refl}) = \mathtt{Id}(A, a, a)$
- The elimination principle is an active constant `J` with $\mathtt{dom}(\mathtt{J}) = \mathtt{Id}(A, a, x)$, $\mathtt{cod}(\mathtt{J}) = P\ x\ e$, and

$$\mathtt{params}(\mathtt{J}) = \mathtt{params}(\mathtt{Id}),\ P :^{\mathbb{ty}} \Pi(x :^{\mathbb{ty}} A)\ \mathtt{Id}(A, a, x) \to \square^{\mathbb{ty}},\ pr :^{\mathbb{ty}} P\ a\ (\mathtt{refl}(A, a))$$

- To define the rewrite rule for `J` we set:

$$\begin{aligned} \Delta_{\mathrm{lin}} &= \mathtt{params}(\mathtt{J}),\ B :^{\mathbb{ty}} \square^{\mathbb{ty}},\ b :^{\mathbb{ty}} B \\ \Delta &= A :^{\mathbb{ty}} \square^{\mathbb{ty}},\ a :^{\mathbb{ty}} A,\ P :^{\mathbb{ty}} \Pi(x :^{\mathbb{ty}} A)\ \mathtt{Id}(A, a, x) \to \square^{\mathbb{ty}},\ pr :^{\mathbb{ty}} P\ a\ (\mathtt{refl}(A, a)) \end{aligned}$$

  Then we can introduce a well-typed rule:

$$\mathtt{Id}, \mathtt{refl}; A, a, a, P, pr, A, a : \Delta \hookrightarrow \Delta_{\mathrm{lin}} \vdash \mathtt{J}(A, a, x, P, pr; \mathtt{refl}(B, b)) \rightsquigarrow pr$$

  Note here that the rewrite rule is highly non-linear: all the endpoints of equalities $a, x, b \in \Delta_{\mathrm{lin}}$ are enforced to coincide through typing.

Using propositional equality, other inductive families can be defined in the so-called "Ford" style [McBride 1999, §3.5], where proper indices are simulated by parameters and equalities. For example, to define vectors, we would have a type family with two parameters $A :^{\mathbb{ty}} \mathbb{ty}, n :^{\mathbb{ty}} \mathbb{N}$ and the two constructors would respectively be guarded by proofs of $\mathtt{Id}(\mathbb{N}, n, \mathtt{0})$ and $\mathtt{Id}(\mathbb{N}, n, \mathtt{S}\ n')$. A more categorically inspired, equivalent presentation of inductive families can be found in [Herbelin and Spiwack 2013, §3.2] using Σ-types and identity types to define proper indexed sums. We expect higher-order recursive types like W-types could also fit in this framework with a more elaborate handling of recursive occurrences ($?^{\mathtt{rec}} n[p]$); we leave this as future work.

Beyond record types, generic coinductive types such as streams defined by co-pattern matching [Abel et al. 2013] are almost within reach of MuTT, in particular the logical relation developed in §5 readily accommodate them. This would however require to dualize the treatment of reduction rules Fig. 5 and their typing (Rule Valid-Rew) to take into account co-recursive occurrences.

3.2 Prop

The Coq proof assistant features a sort $\mathbb{P}$ (`Prop`) of propositions compatible with a proof erasure semantics, a key property for extracting formally verified programs from Coq developments. This compatibility is obtained through a restricted elimination schema from Prop into Type, known as *singleton elimination*, that enforces that $\mathbb{P}$ is compatible with proof-irrelevance but does not impose that axiom upfront. Singleton elimination, first studied explicitly by Letouzey [2004], restricts elimination on inductive types from $\mathbb{P}$ to $\mathbb{ty}$ to those inductives that have syntactically at most one constructor and whose arguments are all in the sort $\mathbb{P}$. As a consequence, the standard proof to distinguish the constructors of an inductive type, e.g. to distinguish `true` from `false` in $\mathbb{B}$, cannot be reproduced for types in $\mathbb{P}$, e.g. $\mathbb{B}_{\mathbb{P}}$: the first step of the proof builds a predicate $P : \mathbb{B}_{\mathbb{P}} \to \square^{\mathbb{ty}}$ such that P `true` is inhabited and P `false` is empty, a step that requires the forbidden elimination of $\mathbb{B}_{\mathbb{P}} : \mathbb{P}$ into the universe $\square^{\mathbb{ty}} : \mathbb{ty}$.

Putting aside the peculiar aspects attached to impredicativity, that indeed turn $\mathbb{P}$ into a hierarchy on its own, it is relatively straightforward to encode a predicative variant of Coq-style $\mathbb{P}$ in MuTT. We consider a fresh sort $\mathbb{P}$, and populate it with inductive type formers $\mathtt{I}(\overline{p})$ and constructors $\mathtt{c}(\overline{x}) : \mathtt{I}(\overline{p})$ as described in §3.1 but restrict the usual eliminators `IRec` to $\mathbb{P}$-valued families, that is predicates $P : \mathtt{I}(\overline{p}) \to \square^{\mathbb{P}}$. Eliminators $\mathtt{singletonRec}_{\mathtt{I}}$ for $\mathbb{ty}$-valued families are added only for inductives $\mathtt{I}(\overline{p})$ that do satisfy the singleton criterion.

3.3 Exceptions

Exceptional Type Theory (ExcTT) of [Pédrot and Tabareau 2018] extends MLTT with the ability to raise exceptions at any type. This theory is further refined in [Pédrot et al. 2019] in order to reason on exceptional terms in a consistent context. This is achieved by introducing two sorts: a sort of pure types embedding standard MLTT and a sort $\mathbb{Exc}$ for the exceptional hierarchy. The use of exceptions is confined to types residing in the exceptional hierarchy, which is explicit in the typing rule for the operator `raise` raising those exceptions:

$$\frac{\Sigma\,;\Gamma \vdash A : \mathbb{Exc}}{\Sigma\,;\Gamma \vdash \mathtt{raise}(A) :^{\mathbb{Exc}} A}$$

The eliminator for inductive types in $\mathbb{Exc}$ must then account for these exceptions, requiring a special catch clause. For instance, the eliminator from exceptional booleans $\mathbb{B}_{\mathbb{Exc}}$ to a sort s takes the following form:

$$\frac{\begin{array}{c}\Sigma\,;\Gamma \vdash P :^{\mathbb{ty}} \mathbb{B}_{\mathbb{Exc}} \to \square^{s} \qquad \Sigma\,;\Gamma \vdash b :^{\mathbb{Exc}} \mathbb{B}_{\mathbb{Exc}} \\ \Sigma\,;\Gamma \vdash h_t :^{s} P\ \mathtt{true} \qquad \Sigma\,;\Gamma \vdash h_f :^{s} P\ \mathtt{false} \qquad \Sigma\,;\Gamma \vdash h_r :^{s} P\ (\mathtt{raise}(\mathbb{B}_{\mathbb{Exc}}))\end{array}}{\Sigma\,;\Gamma \vdash \mathtt{catch}_{\mathbb{B}}(P, h_t, h_f, h_r; b) :^{s} P\ b}$$

The specification of this eliminator is completed with the reduction rules where $\overline{p} = P, b, h_t, h_f, h_r$:

$$\mathtt{catch}_{\mathbb{B}}(\overline{p}; \mathtt{true}) \Rightarrow h_t \qquad \mathtt{catch}_{\mathbb{B}}(\overline{p}; \mathtt{false}) \Rightarrow h_f \qquad \mathtt{catch}_{\mathbb{B}}(\overline{p}; \mathtt{raise}(\mathbb{B}_{\mathbb{Exc}})) \Rightarrow h_r \qquad (1)$$

We can present ExcTT as an instance of MuTT, reusing the sort $\mathbb{ty}$ for pure types plus a fresh sort $\mathbb{Exc} \in \mathbb{S}$ populated with the inert constants $\mathbb{B}_{\mathbb{Exc}}$, `true`, `false`, $\mathtt{exc}_{\mathbb{B}}$ and the active constants $\mathtt{catch}_{\mathbb{B}}$ and `raise`. The exceptional booleans $\mathbb{B}_{\mathbb{Exc}}$ with $\mathrm{cod}(\mathbb{B}_{\mathbb{Exc}}) = \square^{\mathbb{Exc}}$ have as constructors, beside the standard `true` and `false` constructors, a new constructor $\mathtt{exc}_{\mathbb{B}}$, all without parameters. This makes $\mathbb{B}_{\mathbb{Exc}}$ a well-formed positive type in the empty signature and $\mathtt{catch}_{\mathbb{B}}$ is introduced as

an eliminator on this well-formed type with:

$$\mathsf{params}(\mathtt{catch}_{\mathbb{B}}) = P :^{\mathbb{ty}} \mathbb{B}_{\mathbb{Ex C}} \to \square^{s},\ h_t :^{s} P\,\mathtt{true},\ h_f :^{s} P\,\mathtt{false},\ h_r :^{s} P\,\mathtt{exc}_{\mathbb{B}}$$

$$\mathsf{dom}(\mathtt{catch}_{\mathbb{B}}) = \mathbb{B}_{\mathbb{Ex C}} \qquad \mathbb{B}_{\mathbb{Ex C}}\,;\mathsf{params}(\mathtt{catch}_{\mathbb{B}}), b :^{\mathbb{Ex C}} \mathsf{dom}(\mathtt{catch}_{\mathbb{B}}) \vdash \mathsf{cod}(\mathtt{catch}_{\mathbb{B}}) = P\,b : s$$

together with the linear equations presented in (1). Again, $\mathtt{catch}_{\mathbb{B}}$ is well-formed in the signature $\mathbb{B}_{\mathbb{Ex C}}$ because it is deterministic, as all patterns have different head symbols, and satisfies rewrite progress, as all non-neutral weak-head normal forms of type $\mathbb{B}_{\mathbb{Ex C}}$ (*i.e.,* $\mathtt{true}$, $\mathtt{false}$ and $\mathtt{exc}_{\mathbb{B}}$) do react. Also, parameters, domain, codomain and rewrite rules are well-typed with respect to the signature $\mathbb{B}_{\mathbb{Ex C}}$. Finally, $\mathtt{raise}$ is presented as an eliminator with $\mathsf{params}(\mathtt{raise}) = \cdot$, hence abbreviated $\mathtt{raise}(A)$ instead of $\mathtt{raise}(\cdot; A)$, defined on the universe $\mathsf{dom}(\mathtt{raise}) := \square^{\mathbb{Ex C}}$, with codomain $\mathsf{cod}(\mathtt{raise}) = A, \mathbb{S}^{\mathsf{cod}}(\mathtt{raise}) = \mathbb{Ex C}$ where $A :^{\mathbb{ty}} \square^{\mathbb{Ex C}}$ is the variable provided by the domain. $\mathtt{raise}$ then comes with a rewrite rule for Π-types and another one for booleans:

$$(\cdot, \mathtt{raise}(\mathbb{B}_{\mathbb{Ex C}}), \mathtt{exc}_{\mathbb{B}}) \qquad ((A :^{\mathbb{ty}} \square^{s_1},\ B :^{\mathbb{ty}} A \to \square^{\mathbb{Ex C}}), \mathtt{raise}(\Pi^{s_1}\,A\,(?^{\mathsf{rec}}B[!])), B^{\mathsf{rec}})$$

The well-formedness of $\mathtt{raise}$ is established as follows. Determinism comes from the fact that all rewrite rules have a distinct head symbol in their pattern. All non-neutral weak-head normal forms in $\square^{\mathbb{Ex C}}$ ($\mathbb{B}_{\mathbb{Ex C}}$ and $\Pi\,A\,B$) do react. The parameters, domain and codomain of $\mathtt{raise}$ are well-typed. Regarding the typing of the rewrite rules, the first one is easily well-typed. As for the second, the Π pattern is indeed well-typed given the context, and the variable B^{rec} comes from $\mathtt{rec}_{\mathtt{raise}}(\Pi^{s_1}\,A?^{\mathsf{rec}}B[!])$ and has type $\Pi(y :^{s_1} A)(B\,y)$.

Note that, if we wanted to extend a base signature with additional type constructors from §3.1 like Σ-types or identity types, the framework would require to consider these type constructors as additional inert constants on which $\mathtt{raise}$ should react.

3.4 Axioms, locally

By parametrizing adequately MuTT, it is possible to add and work with a new axiom inhabiting any chosen type Ax without compromising the canonicity of $\mathbb{ty}$. This is achieved by creating a new isolated sort $\mathbb{Ax}$ which is a fresh copy of $\mathbb{ty}$, except that eliminations are restricted to $\mathbb{Ax}$. Then, the axiom can be realized by adding an active term $\mathtt{axiom} :^{\mathbb{Ax}} Ax$ with no parameters ($\mathsf{params}(\mathtt{axiom}) = \cdot$) and $\mathsf{dom}(d) = \mathtt{Unit}$, the trivial inductive type (or equivalently, no argument at all). We do not attach any rewrite rule to $\mathtt{axiom}$ as this would amount to realizing the axiom itself, breaking progress and therefore, $\mathtt{axiom}$ is well-formed only because $\mathbb{Ax}$ is isolated, which precisely prevents leaking the axiom into $\mathbb{ty}$.

However, the isolation property does not prevent us from defining a boxing mechanism from $\mathbb{ty}$ into $\mathbb{Ax}$ with elimination into $\mathbb{Ax}$ that allows us to prove properties on inhabitants of $\mathbb{ty}$ using $\mathtt{axiom}$, but only in the axiomatic sort $\mathbb{Ax}$. Then, depending on the design choice, one can define one axiomatic sort per axiom to encapsulate clearly which axiom has been used directly in the type information, or consider an axiomatic sort where any axiom can be postulated, thus encapsulating in the types that an unsafe version of $\mathbb{ty}$ has been used.

This provides a type-theoretic, local and modular alternative to the `--safe` pragma of Agda, or the `Print Assumption` checker of Coq. Also, it allows users to make use of several incompatible axioms in the same development, as long as they are postulated in different isolated sorts.

3.5 Dependent elimination through universe unboxing

A sort $s \in \mathbb{S}$ has booleans if it is equipped with a type $\mathbb{B}_s : s$, terms $\mathtt{true}_s, \mathtt{false}_s :^{s} \mathbb{B}_s$ and an induction principle

$$\mathtt{ind}_{\mathbb{B}_s} : (P :^{\mathbb{ty}} \mathbb{B}_s \to \square^{s})(p_t :^{s} P\,\mathtt{true}_s)(p_f :^{s} P\,\mathtt{false}_s)(b :^{s} \mathbb{B}_s) \to P\,b.$$

As explained in §3.2, this data is however not enough to show expected properties of booleans, for instance to derive that $\mathtt{true}_s \not\equiv \mathtt{false}_s$. In order to recover the full power of large elimination on booleans of sort s, we need the ability to define predicates taking value in $\Box^s$ by case analysis:

$$\frac{\Sigma\,;\Gamma \vdash P_t :^{\mathbb{ty}} \Box^s \qquad \Sigma\,;\Gamma \vdash P_f :^{\mathbb{ty}} \Box^s \qquad \Sigma\,;\Gamma \vdash b :^s \mathbb{B}_s}{\Sigma\,;\Gamma \vdash \mathtt{ind}^{\Box^s}_{\mathbb{B}_s}\, P_t\, P_f\, b :^{\mathbb{ty}} \Box^s}$$

Note that this induction principle $\mathtt{ind}^{\Box^s}_{\mathbb{B}_s}$ specialized to $\Box^s$ is not an instance of $\mathtt{ind}_{\mathbb{B}_s}$ because $\Box^s$ resides in the sort $\mathbb{ty}$.

Rather than requiring for each inductive type in sort s to come equipped with two elimination principles, it is actually enough to have a reflection of $\Box^s$ in sort s, that is a type $\mathbb{B}\mathrm{ox}\,\Box^s : s$ equipped with terms $\Sigma\,;\vdash \mathtt{box} :^s \Box^s \to \mathbb{B}\mathrm{ox}\,\Box^s$ and $\Sigma\,;\vdash \mathtt{unbox} :^{\mathbb{ty}} \mathbb{B}\mathrm{ox}\,\Box^s \to \Box^s$ such that $\Sigma\,;\Gamma \vdash \mathtt{unbox}\,(\mathtt{box}\,A) \equiv A :^{\mathbb{ty}} \Box^s$ for any type $A :^{\mathbb{ty}} \Box^s$. Given such a reflection, we can derive the induction principle $\mathtt{ind}^{\Box^s}_{\mathbb{B}_s}$ from the standard induction principle as follows:

$$\mathtt{ind}^{\Box^s}_{\mathbb{B}_s}\, P_t\, P_f\, b \overset{\text{def}}{=} \mathtt{unbox}\left(\mathtt{ind}_{\mathbb{B}_s}(\lambda(x : \mathbb{B}_s).\, \mathbb{B}\mathrm{ox}\,\Box^s)\ (\mathtt{box}\,P_t)\ (\mathtt{box}\,P_f)\ b\right) :^{\mathbb{ty}} \Box^s$$

Of course, such a reflection does not always exist for an arbitrary sort s, in particular in the case of $\mathbb{P}$. But it exists for instance for $\mathbb{E}\mathbb{x}\mathbb{c}$ which justifies why dependent elimination is valid in ExcTT. The term $\mathbb{B}\mathrm{ox}\,\Box^{\mathbb{E}\mathbb{x}\mathbb{c}} : \mathbb{E}\mathbb{x}\mathbb{c}$ is basically obtained as $\Box^{\mathbb{E}\mathbb{x}\mathbb{c}}$ plus a default type ✠ in $\mathbb{E}\mathbb{x}\mathbb{c}$ for the exception in $\mathbb{B}\mathrm{ox}\,\Box^{\mathbb{E}\mathbb{x}\mathbb{c}}$, *i.e.*, $\mathtt{raise}(\mathbb{B}\mathrm{ox}\,\Box^{\mathbb{E}\mathbb{x}\mathbb{c}}) \equiv ✠$.

4 MODULARITY OF MuTT

As formalized and illustrated previously, MuTT is extensible by means of its parametrization. We now show that MuTT delivers on the *modularity* front: two parametrizations of MuTT can be merged seamlessly, preserving the metatheoretical results of each independent parametrization. As explained in the introduction, modularity is key to allow developments to locally rely on extensions such as exceptions or axioms, without interfering with each other, i.e. the metatheoretical properties in §5 are always preserved.

A parametrization $\mathcal{P}' = (\mathbb{S}', \Sigma')$ is a proper extension of $\mathcal{P} = (\mathbb{S}, \Sigma)$, noted $\mathcal{P} \rightarrowtail \mathcal{P}'$, when $\mathbb{S} \subseteq \mathbb{S}'$, each component of Σ are in Σ', isolation is preserved and any active constant $d \in \Sigma'$ defined on a universe s is either already in Σ or occurs on a sort not appearing in $\mathbb{S}$:

$$\forall d \in \Sigma', \qquad \mathrm{dom}(d) = \Box^s \quad \Rightarrow \quad d \in \Sigma \vee s \notin \mathbb{S}$$

For any parametrization $\mathcal{P}$, the identity is a proper extension $\mathcal{P} \rightarrowtail \mathcal{P}$ and proper extensions compose, forming a preorder with initial object $\mathcal{P}_{\mathbb{ty}} = (\{\mathbb{ty}\}, \cdot)$.

Lemma 8 (Functoriality). *All typing judgments of* MuTT *presented in Fig. 1 are functorial with respect to proper extensions, that is, if* $(\mathbb{S}, \Sigma) \rightarrowtail (\mathbb{S}', \Sigma')$ *and* $\Sigma; \Gamma \vdash \mathcal{J}$ *is a derivable judgment then* $\Sigma'; \Gamma \vdash \mathcal{J}$ *is also derivable.*

Proof. By induction on the derivation of the judgment $\Sigma; \Gamma \vdash \mathcal{J}$ noting that typing derivations only use that constants belongs to Σ, a property preserved by proper extensions. □

Importantly, well-formed signature extensions, as defined in Definitions 4 to 7, are compatible with proper extensions, with the exception of active constants defined on universes:

Lemma 9. *Suppose* $(\mathbb{S}, \Sigma) \rightarrowtail (\mathbb{S}', \Sigma')$.

Positive type extension *If* $\Sigma, (K, \mathcal{I}_K)$ *is a well-formed signature then so is* $\Sigma', (K, \mathcal{I}_K)$*;*
Negative type extension *If* $\Sigma, (K, \mathcal{A}_K)$ *is a well-formed signature then so is* $\Sigma', (K, \mathcal{A}_K)$*;*
Eliminator extension *If* $\Sigma, (d, \mathcal{R}_d)$ *is a well-formed signature with* $\mathrm{dom}(d) = K(\overline{u})$ *then so is* $\Sigma', (d, \mathcal{R}_d)$*;*

Builder extension *If* $\Sigma, (c, \mathcal{R}_c)$ *is a well-formed signature then so is* $\Sigma', (c, \mathcal{R}_c)$.

Proof. Suppose $(K, \mathcal{I}_K)$ is a well-formed positive type in Σ with $\mathrm{cod}(K) = \square^s$, we show that it is a well-formed positive type in Σ'. By functoriality of judgments, all conditions but the first are satisfied. Suppose $d \in \Sigma'$ has domain a universe $\square^{s'}$ then either $d \in \Sigma$ and, by well-formedness of K, $s' \neq s$, or $s' \notin \mathbb{S}$ so $s' \neq s$. A similar argument applies for well-formed negative types.

Suppose $(d, \mathcal{R}_d)$ is a well-formed eliminator on a positive constant, $\mathrm{dom}(d) = K(\overline{u})$ in Σ, we show that it is well-formed as well in Σ'. Since $(K, \mathcal{I}_K) \in \Sigma$ and proper extensions preserve components, $(K, \mathcal{I}_K) \in \Sigma'$. $\mathrm{det}(\mathcal{R}_d)$ and $\mathrm{progress}(\mathcal{I}_K, d, \mathcal{R}_d)$ are independent from the signature and all the typing conditions are consequences of functoriality along proper extensions. If $\mathbb{S}^{\mathrm{dom}}(d)$ is isolated in Σ then $\mathbb{S}^{\mathrm{cod}}(K)$ is isolated in Σ, and they are both isolated in Σ' because proper extensions preserve isolation. Again, a similar argument applies for well-formed builders. □

Theorem 10 (Combining parametrizations). *Let* $\mathcal{P} = (\mathbb{S}, \Sigma)$ *be a parametrization of* MuTT *and* $\mathcal{P}_1 = (\mathbb{S}_1, \Sigma_1)$, $\mathcal{P}_2 = (\mathbb{S}_2, \Sigma_2)$ *two proper extensions of* $\mathcal{P}$. *There exists a well-formed signature* $\Sigma_\cup$ *on* $\mathbb{S}_\cup = \mathbb{S}_1 \uplus_{\mathbb{S}} \mathbb{S}_2$ *such that* $\mathcal{P}_\cup = (\mathbb{S}_\cup, \Sigma_\cup)$ *is a proper extension of both* $\mathcal{P}_1$ *and* $\mathcal{P}_2$.

Proof. Without loss of generality, we can assume that Σ is a prefix of $\Sigma_1 = \Sigma\Sigma'_1$. The construction of $\Sigma_\cup$ then proceeds by induction on Σ'_1, extending inductively the proper extension $\mathcal{P} \rightarrowtail \mathcal{P}_2$. All cases are covered by Lemma 9, except for the case of a well-formed active constant with domain a universe, which we now discuss. Assume that $(d, \mathcal{R}_d)$ is well-formed in $\Sigma\Sigma'_1$, $\mathrm{dom}(d) = \square^s$. The induction hypothesis states that we have well-formed signatures $(\mathbb{S}_1, \Sigma\Sigma'_1) \rightarrowtail (\mathbb{S}_\cup, \Sigma_2\Sigma'_1)$ together with the indicated proper extension. Since $d \in \Sigma_1$ is not part of the prefix Σ, by properness of $\mathcal{P} \rightarrowtail \mathcal{P}_1$, $s \notin \mathbb{S}$ so $s \notin \mathbb{S}_2$ and the signatures $\Sigma\Sigma'_1$ and $\Sigma_2\Sigma'_1$ introduce the same constants in the universe $\square^s$. Therefore, if $\mathrm{progress}(\mathcal{I}_d, d, \mathcal{R}_d)$ holds for $\mathcal{I}_d$ computed in $\Sigma\Sigma'_1$ it also holds for $\mathcal{I}_d$ computed in $\Sigma_2\Sigma'_1$. The other conditions to show that $(d, \mathcal{R}_d)$ is well formed in $\Sigma_2\Sigma'_1$ hold by functoriality of typing judgments, preservation of isolation and independence with respect to the signature for $\mathrm{det}(\mathcal{R}_d)$ thus concluding the inductive step. By construction, we have both $(\mathbb{S}_2, \Sigma_2) \rightarrowtail (\mathbb{S}_\cup, \Sigma_\cup)$ and $(\mathbb{S}_1, \Sigma_1) \rightarrowtail (\mathbb{S}_\cup, \Sigma_\cup)$. □

As a crude application of Theorem 10, we can combine almost disjoint parametrizations that agree on the sort $\mathbb{ty}$:

Corollary 11. *If* $\mathcal{P}_1 = (\mathbb{S}_1, \Sigma_{\mathbb{ty}}\Sigma_1)$ *and* $\mathcal{P}_2 = (\mathbb{S}_2, \Sigma_{\mathbb{ty}}\Sigma_2)$ *are* MuTT *parametrization that only share* $\mathbb{S}_1 \cap \mathbb{S}_2 = \{\mathbb{ty}\}$, *and agree on a common signature* $(\{\mathbb{ty}\}, \Sigma_{\mathbb{ty}})$, *then* $(\mathbb{S}_1 \cup \mathbb{S}_2, \Sigma_{\mathbb{ty}}\Sigma_1\Sigma_2)$ *is a valid* MuTT *parametrization.*

In other words, combined with the metatheoretical results presented in § 2.4, this corollary shows that MuTT addresses the logical modularity issue depicted in the introduction of this paper. Combining two valid MuTT parametrizations yields to a consistent type theory, even if the logical principles provided in those theories are not compatible altogether. This logical frontier has been achieved by the multiverse setting that allows to localize in a sort the use of new logical principles, and also the use of their consequences, which may not even mention explicitly those new principles.

5 METATHEORY OF MuTT

In this section, we show the metatheoretical properties of MuTT claimed in § 2 by adapting and extending the mechanized logical relation proof of Abel et al. [2018]. The high-level idea of the proof is standard: we carefully define a logical relation exhibiting for each derivable judgments of MuTT a canonical standard shape for its derivation, show that the resulting logical relation satisfies a variety of properties then prove the fundamental lemma by induction on typing derivations. Finally we derive the actual metatheoretical properties as consequences of the fundamental lemma.

$\Sigma\,;\Gamma \Vdash \Delta$	Δ is a reducible telescope on top of Γ with respect to signature Σ
$\Sigma\,;\Gamma \Vdash \sigma : \Delta \mid [\Delta]$	σ is a reducible substitution to the extension $[\Delta] : \Sigma\,;\Gamma \Vdash \Delta$
$\Sigma\,;\Gamma \Vdash A : s$	A is a reducible type at sort $s \in \mathbb{S}$ in context Γ
$\Sigma\,;\Gamma \Vdash t :^s A \mid [A]$	t is a reducible term of reducible type $[A] : \Sigma\,;\Gamma \Vdash A : s$
$\Sigma\,;\Gamma \Vdash A \mid [A] \equiv B : s$	B is convertible to the reducible type $[A] : \Sigma\,;\Gamma \Vdash A : s$
$\Sigma\,;\Gamma \Vdash t \equiv u :^s A \mid [A]$	t and u are convertible at reducible type $[A] : \Sigma\,;\Gamma \Vdash A : s$
$\Sigma;\Gamma \Vdash_{\mathrm{ne}} t :^s A$	t is a reducible neutral term of type A

Fig. 7. Components of the logical relation for MuTT

5.1 Logical relation

The logical relation defines families of types, the *reducibility* relations $\Vdash$ described in Fig. 7, corresponding to each judgment of MuTT (Fig. 1). The definition of these relation proceed first by induction on the well-formed signature Σ, collecting inductively proof of reducibility data employed to define the reducibility relation at types introduced by inert constants K (Definitions 4 and 5). We use the notation $[x]$ for the proof of reducibility associated to a type, term, context or substitution x. We abuse application notation $[A]\ [t]$ to substitute a reducibility proof $[t] : \Sigma\,;\Gamma \Vdash t :^s X \mid [X]$ in $[A] : \Sigma;\Gamma, x :^s X \Vdash \mathcal{J}$, omitting the required substitutions and providing only the main arguments.

Definition 8 (Reducibility of positive type constant). *A positive type constant $(K, \mathcal{I}_K)$ is reducible in signature Σ if*

(1) its parameters are reducible $[\mathrm{params}(K)] : \Sigma\,;\cdot \Vdash \mathrm{params}(K)$

(2) for each inert constant $c \in \mathcal{I}_K$, $\mathrm{cod}(c) = K(\overline{u})$, $\mathrm{params}(c)$, $\mathrm{dom}(c)$ and $\overline{u}$ are reducible

$$[\mathrm{params}(c)] : \Sigma\,;\cdot \Vdash \mathrm{params}(c) \qquad [\overline{u}] : \Sigma\,;\mathrm{params}(c) \Vdash \overline{u} : \mathrm{params}(K) \mid [\mathrm{params}(K)]$$

$$\forall i,\quad \mathrm{dom}(c)_i = K(\overline{t}) \wedge [\mathrm{dom}(c)_i] : \Sigma\,;\mathrm{params}(c) \Vdash \overline{t} : \mathrm{params}(K) \mid [\mathrm{params}(K)]$$

Definition 9 (Reducibility of negative type constant). *A negative type constant $(K, \mathcal{A}_K)$ is reducible in signature Σ if*

(1) its parameters are reducible $[\mathrm{params}(K)] : \Sigma\,;\cdot \Vdash \mathrm{params}(K)$

(2) for each active constant $d_i \in \mathcal{A}_K = \overline{d},$, $\mathrm{cod}(d_i)$ is reducible

$$\begin{cases} [\mathrm{cod}(d_i)] : \Sigma\,;\mathrm{params}(K), \overline{\mathrm{cod}(d)}_{<i} \Vdash \overline{t} : \mathrm{params}(K) \mid [\mathrm{params}(K)] & \text{if } \mathrm{cod}(d_i) = K(\overline{t}) \\ [\mathrm{cod}(d_i)] : \Sigma\,;\mathrm{params}(d), \overline{\mathrm{cod}(d)}_{<i} \Vdash \mathrm{cod}(d_i) : \mathbb{S}^{\mathrm{cod}}(d_i) & \text{otherwise} \end{cases}$$

Definition 10 (Reducibility of rewrite rules). *A rewrite rule $(\sigma : \Delta \hookrightarrow \Delta_{\mathrm{lin}}, d(\overline{x}; pat), r)$ is reducible in signature Σ if its contexts, substitutions and right hand side are reducible:*

$$[\Delta] : \Sigma\,;\cdot \Vdash \Delta \qquad [\Delta_{\mathrm{lin}}] : \Sigma\,;\cdot \Vdash \Delta_{\mathrm{lin}} \qquad \Sigma\,;\Delta_{\mathrm{lin}} \Vdash \mathtt{rec}_d(pat) \qquad \Sigma\,;\Delta \Vdash \sigma : \Delta_{\mathrm{lin}} \mid [\Delta_{\mathrm{lin}}]$$

$$\Sigma\,;\Delta_{\mathrm{lin}} \Vdash \overline{x} : \mathrm{params}(d) \mid [\mathrm{params}(d)] \qquad \Sigma\,;\Delta_{\mathrm{lin}} \Vdash \epsilon(pat) :^{\mathbb{S}^{\mathrm{dom}}(d)} A \mid [A]$$

$$\Sigma\,;\Delta, \mathtt{rec}_d(pat)[\sigma] \Vdash r :^{\mathbb{S}^{\mathrm{cod}}(d)} \mathrm{cod}(d)[\overline{x}, \epsilon(pat)][\sigma] \mid [\mathrm{cod}(d)]\ \ldots$$

Assuming by induction hypothesis that we have the reducibility datum $[\Sigma]$ corresponding to a well-formed signature Σ, we can now describe the defining cases of the logical relation. In all cases, the general methodology is to first reduce the subject of the judgment to a whnf, and then characterize the canonical forms at each type and judgments. The definition is complicated by two

aspects exposed in [Abel et al. 2018]: first, universes introduce a seemingly circular definition of reducibility at terms and types; second, negative occurrences of the reducibility relation appears in the definition, e.g. for reducibility of terms at dependent products. The circularity induced by universes actually disappears once we take into account the (implicit) universe levels. The problem induced by negative occurrences is solved by defining inductively the reducibility relation on types, and then defining the other relations by recursion on the proofs of reducibility on types. Since the definition of all these relations are mutual, we obtain a well-founded albeit complex inductive recursive definition. Figure 9 describes the inductive part of reducibility of types with a case for universes of each sorts, neutral types, Π-types that use the context reducibility from Fig. 8 and constant introduced from the signature Σ. Figure 10 then dispatches reducibility of terms to auxiliaries definitions according to the reducibility proof of its type. We omit most of these auxiliary definitions, focusing on the components proper to MuTT and absent from [Abel et al. 2018]. Figure 11 describes the reducibility of terms at types introduced with a constant K drawn from the signature Σ. The rule Constant-Reducible is the entry point and closes the other judgments by weak head reduction, while the otherrules apply depending on the positive or negative character of the type K as described by the signature Σ. When $(K, \mathcal{I}_K)$ is a positive type constant according to Σ, a whnf is reducible at $K(\overline{a})$, if it is neutral using Neutral-Reducible, or if it consists of an inert constant $c \in \mathcal{I}_K$, one of the canonical introduction form of K, its parameters and recursive arguments are **inductively** reducible and the arguments of K computed from its parameters are convertible to $\overline{a}$, using rule Inert-Constant-Positive-Reducible. When $(K, \mathcal{A}_K)$ is a negative type constant according to Σ, a whnf t is reducible at $K(\overline{a})$, rule Inert-Constant-Negative-Reducible, if all its possible observations $d(\overline{a}, t)$ for $d \in \mathcal{A}_K$ are **coinductively** reducible at their corresponding type. In both of these cases, the definitions make a crucial use of the reducibility data $[\Sigma]$ obtained by induction hypothesis on Σ.

The so-defined logical relation verifies a handful of properties:

(1) it is stable under weakening, substitution by reducible substitution;
(2) the relations induced by conversion are reflexive, symmetric, transitive and congruent with respect to all type and term formers;
(3) all reducibility relations are stable by judgmental conversion;
(4) a reducible type or term reduces to a whnf that is itself reducible;
(5) reducibility is closed by anti-reduction;
(6) well-typed neutrals are reducible.

We highlight two key properties: the reducibility relations are *irrelevant*, so that being reducible is a mere property; and all judgments satisfy the so called escape lemma that allows to recover derivability of a MuTT judgment out of its reducible counterpart.

Lemma 12 (Irrelevance). *If* $[A], [A'] : \Sigma ; \Gamma \Vdash A : s$ *are two proofs of reducibility of the type* A, $\Sigma ; \Gamma \Vdash t :^s A \mid [A] \Rightarrow \Sigma ; \Gamma \Vdash t :^s A \mid [A']$.

The key property of MuTT needed to prove irrelevance of the reducibility witnesses is the determinism of rewrite rules, ensuring uniqueness of the head of a weak head normal form.

Lemma 13 (Escape). *For any judgment form* $\mathcal{J}$ *of* MuTT, *if* $\Sigma; \Gamma \Vdash \mathcal{J}$ *then* $\Sigma; \Gamma \vdash \mathcal{J}$.

The escape lemma reconstructs a canonical derivation of a judgment out of a reducibility proof. Irrelevance is used pervasively to "realign" reducibility judgments that only differ in the reducibility proof.

$\boxed{\Sigma\,;\Gamma \Vdash \Delta}$ *reducibility of telescopes.*

$$\frac{}{\Sigma\,;\Gamma \Vdash \cdot} \qquad \frac{[\Delta] : \Sigma\,;\Gamma \Vdash \Delta \qquad \Sigma\,;\Gamma, \Delta \mid [\Delta] \Vdash A : s}{\Sigma\,;\Gamma \Vdash \Delta, x :^s A}$$

$$\Sigma\,;\Gamma, \Delta \mid [\Delta] \Vdash A : s \stackrel{\text{def}}{=} \forall \Gamma' \supseteq \Gamma,\ \Sigma\,;\Gamma' \Vdash \sigma : \Delta \mid [\Delta] \to ([A\sigma] : \Sigma\,;\Gamma' \Vdash A[\sigma] : s \ \wedge (\Sigma\,;\Gamma' \Vdash \sigma' : \Delta \mid [\Delta] \to \ \Sigma\,;\Gamma' \Vdash \sigma \equiv \sigma' : \Delta \mid [\Delta] \to \Sigma\,;\Gamma' \Vdash A[\sigma] \mid [A\sigma] \equiv A[\sigma'] : s))$$

$$\Delta \supseteq \Gamma \stackrel{\text{def}}{=} \exists \rho, \Delta \vdash \rho\Gamma \wedge \rho \text{ is a monotone renaming}$$

$\boxed{\Sigma\,;\Gamma \Vdash \sigma : \Delta \mid [\Delta]}$ *reducibility of substitutions.*

$$\frac{}{\Sigma\,;\Gamma \Vdash\, ! : \cdot \mid [\cdot]} \qquad \frac{[\sigma] : \Sigma\,;\Gamma \Vdash \sigma : \Delta \mid [\Delta] \qquad \Sigma\,;\Gamma \Vdash t :^s A[\sigma] \mid ([A]\ id_\Gamma\ [\sigma]).1}{\Sigma\,;\Gamma \Vdash (\sigma, t) : \Delta, x :^s A \mid [\Delta], [A]}$$

Fig. 8. Reducibility of context and substitutions

Univ-Red-Type
$$\frac{\Sigma\,;\Gamma \vdash A \Rightarrow \square^s : \mathbb{ty}}{[\square^s] : \Sigma\,;\Gamma \Vdash A : \mathbb{ty}}$$

Neutral-Red-Type
$$\frac{\Sigma\,;\Gamma \vdash A \Rightarrow T : \mathbb{ty} \qquad \text{neutral}\, T \qquad \Sigma\,;\Gamma \vdash T \equiv T : \square^s \mathbb{ty}}{[\text{ne}] : \Sigma\,;\Gamma \Vdash A : s}$$

Π-Red-Type
$$\frac{\Sigma\,;\Gamma \vdash A \Rightarrow \Pi\,(x :^{s_1} X)\ Y : s_2 \qquad \Sigma\,;\Gamma \vdash X : s_1 \qquad \Sigma\,;\Gamma, x :^{s_1} X \vdash Y : s_2 \qquad \Sigma\,;\Gamma \vdash \Pi\,(x :^{s_1} X)\ Y \equiv \Pi\,(x :^{s_1} X)\ Y : s_2 \qquad \Sigma\,;\Gamma \Vdash x :^{s_1} X, y :^{s_2} Y}{[\Pi] : \Sigma\,;\Gamma \Vdash A : s}$$

Red-Type
$$\frac{\Sigma\,;\Gamma \vdash A \Rightarrow K(\bar{t}) : s \qquad K \in \Sigma \qquad \Sigma\,;\Gamma \vdash \bar{t} \equiv \bar{t} : \text{params}(K) \qquad \Sigma\,;\Gamma \Vdash \bar{t} : \text{params}(K) \mid [\text{params}(K)]}{[\text{Cst}] : \Sigma\,;\Gamma \Vdash A : s}$$

Fig. 9. Reducibility for types

$$\begin{array}{lcll}
\Sigma\,;\Gamma \Vdash t :^{\mathbb{ty}} A \mid [\square^s] & := & \Sigma\,;\Gamma \Vdash t : s & \text{(universe level decreases)} \\
\Sigma\,;\Gamma \Vdash t :^s A \mid [\text{ne}] & := & \Sigma;\Gamma \Vdash_{\text{ne}} t :^s A & \\
\Sigma\,;\Gamma \Vdash t :^s A \mid [\Pi]\ \ldots & := & \text{(omitted)} & \\
\Sigma\,;\Gamma \Vdash t :^s A \mid [\text{Cst}]\ K\ \bar{a}\ \ldots & := & \Sigma;\Gamma \Vdash_K t :^s K(\bar{a}) &
\end{array}$$

Fig. 10. Reducibility for terms

5.2 Fundamental lemma

At a high level, the fundamental lemma states that derivable judgments are valid. More precisely, it consists of a family of lemmas for each judgments of MuTT:

Theorem 14 (Fundamental lemma). *Let Σ be a well-formed signature.*

(1) If $\Sigma\,;\Gamma \vdash$ then $\Sigma\,;\Gamma \Vdash$;
(2) If $\Sigma\,;\Gamma \vdash A : s$ then there is a proof $[A] : \Sigma\,;\Gamma \Vdash A : s$;
(3) If $\Sigma\,;\Gamma \vdash t :^s A$ then $\Sigma\,;\Gamma \Vdash t :^s A \mid [A]$
(4) If $\Sigma\,;\Gamma \vdash A \equiv B : s$ then $\Sigma\,;\Gamma \Vdash A \mid [A] \equiv B : s$
(5) If $\Sigma\,;\Gamma \vdash t \equiv u :^s A$ then $\Sigma\,;\Gamma \Vdash t \equiv u :^s A \mid [A]$

$$\textsc{Constant-Reducible}\quad \frac{\Sigma\,;\Gamma \vdash t \Rightarrow w :^s K(\overline{a}) \qquad \Sigma;\Gamma \vdash w \cong w :^s K(\overline{a}) \qquad \Sigma;\Gamma \Vdash^{\mathrm{nf}}_K w :^s K(\overline{a})}{\Sigma;\Gamma \Vdash_K t :^s K(\overline{a})}$$

$$\textsc{Neutral-Reducible}\quad \frac{\Sigma;\Gamma \Vdash_{\mathrm{ne}} n :^s K(\overline{a})}{\Sigma;\Gamma \Vdash^{\mathrm{nf}}_K n :^s K(\overline{a})}$$

$$\textsc{Inert-Constant-Positive-Reducible}\quad \frac{\begin{array}{c}(K, \mathcal{I}_K) \in \Sigma \qquad c \in \mathcal{I}_K \qquad \Sigma\,;\Gamma \Vdash \overline{p} : \mathrm{params}(c) \mid [\mathrm{params}(c)] \\ \forall i, [v_i] : \Sigma;\Gamma \Vdash_K v_i :^s \mathrm{dom}(c)_i[\overline{p}] \qquad \mathrm{cod}(c) = K(\overline{u}_c) \qquad \Sigma;\Gamma \vdash \overline{u}_c[\overline{p}] \cong \overline{a} : \mathrm{params}(K)\end{array}}{\Sigma;\Gamma \Vdash^{\mathrm{nf}}_K c(\overline{p}, \overline{v}) :^s K(\overline{a})}$$

$$\textsc{Inert-Constant-Negative-Reducible}\quad \frac{\begin{array}{c}(K, \mathcal{A}_K) \in \Sigma, \quad \mathcal{A}_K = \overline{d} \\ \forall i, \begin{cases} [d]_i : \Sigma;\Gamma \Vdash^{\mathrm{nf}}_K d_i(\overline{a}, t) :^s K(\overline{t}[\overline{a}, \overline{d(\overline{a}, t)}_{<i}]) & \text{if } \mathrm{cod}(d_i) = K(\overline{t}) \\ [d]_i : \Sigma\,;\Gamma \Vdash d_i(\overline{a}, t) :^{\mathbb{S}^{\mathrm{cod}}(d_i)} \mathrm{cod}(d_i)[\overline{a}, \overline{d(\overline{a}, t)}_{<i}] \mid [\mathrm{cod}(d_i)]\ [\overline{a}]\ \overline{[d]}_{<i} & \text{otherwise} \end{cases}\end{array}}{\Sigma;\Gamma \Vdash^{\mathrm{nf}}_K t :^s K(\overline{a})}$$

Fig. 11. Reducibility of terms at constant types

The proof of the fundamental lemma proceed by induction on the typing derivation, generalizing the result to be proved by uniformly closing reducibility under substitution and extensionality (see the definition of $\Sigma\,;\Gamma, \Delta \mid [\Delta] \Vdash A : s$ in Fig. 8) and proving the result mutually for all judgments. The case of reducibility of universes, constant types, introduction of inert constants for positive types and introduction of active constants for negative types are mostly straightforward: we organized the logical relation so that there is already a case available for these forms. The challenging and interesting cases are thus the dual ones that are not explicitly mentioned in the logical relation: the introduction of active constants for positive types and universes and the introduction of inert constants for negative types. We sketch the proof for the first case, highlighting some required properties of MuTT participating to its design.

Proof. Consider a typing derivation ending with the rule Active-Term, with conclusion $\Sigma\,;\Gamma \vdash d(\overline{t}, u) :^{\mathbb{S}^{\mathrm{cod}}(d)} \mathrm{cod}(d)[\overline{t}, u]$ where $\mathrm{cod}(d) = K(\overline{u_d})$ and $(K, \mathcal{I}_K)$ is a positive type according to Σ. By induction hypothesis, we have that $d \in \Sigma$, so that the parameters, domain and codomain of d are reducible and

$$[\overline{t}] : \Sigma\,;\Gamma \Vdash \overline{t} : \mathrm{params}(d) \mid [\mathrm{params}(d)], \qquad [u] : \Sigma\,;\Gamma \Vdash u :^{\mathbb{S}^{\mathrm{dom}}(d)} \mathrm{dom}(d)[\overline{t}] \mid [\mathrm{dom}(d)]\ [\overline{t}]$$

Since u is reducible, it reduces to a weak head normal form w, reducible at the same type, that is $[w] : \Sigma;\Gamma \Vdash_K w :^{\mathbb{S}^{\mathrm{dom}}(d)} K(\overline{u_d}[\overline{t}])$. By anti-reduction, it is enough to show that $d(\overline{t}; w)$ is reducible, which we do by induction on $[w]$, generalizing over the reducible parameters $\overline{t}$. By inversion, $[w]$ is necessarily produced with an instance of Constant-Reducible, and is either a neutral or of the shape $w = c(\overline{p}, \overline{v})$ for $c \in \mathcal{I}_K$ (by Inert-Constant-Positive-Reducible). If w is neutral or $\neg\mathrm{react}(d, c, \mathcal{R})$ holds, then $d(\overline{t}; w)$ is neutral, can be shown to be well-typed using the escape lemma, so it is reducible. Otherwise, $\mathrm{react}(d, c, \mathcal{R})$ ensures that there exist a rewrite rule $(\sigma : \Delta \hookrightarrow \Delta_{\mathrm{lin}}, d(\overline{x}; pat), r) \in \mathcal{R}$ such that $d(\overline{x}; pat)$ unifies with $d(\overline{t}; w)$ thanks to linearity, yielding a reducible substitution $\Sigma\,;\Gamma \Vdash \rho : \Delta_{\mathrm{lin}} \mid [\Delta_{\mathrm{lin}}]$. Using the premise of Inert-Constant-Positive-Reducible obtained from $[w]$, we have that $w = c(\overline{p}, \overline{v})$, $\mathrm{cod}(c) = K(\overline{u_c})$ and $\Sigma\,;\Gamma \vdash \overline{u_c}[\overline{p}] \equiv \overline{u_d}[\overline{t}] : \mathrm{params}(K)$. By linearizability of the rewrite rule (Definition 1), we obtain a renaming τ that is an inverse of σ up to conversion. The composed substitution $\tau[\rho]$ is reducible, $\Sigma\,;\Gamma \Vdash \nu[\rho] : \Delta \mid [\Gamma][\Delta]$, because ρ is reducible and τ is a renaming. By substitution into the reducibility proof of the right hand side r obtained from the reducibility of the signature $[\Sigma]$

together with the induction hypothesis on $[w]$ for recursive occurrences from the pattern, we have that $r[\tau[\rho], \rho^{\mathsf{rec}}{}_d(pat)[\tau[\rho]]]$ is reducible at type

$$\mathrm{cod}(d)[\overline{x}, \epsilon(pat)][\sigma][\tau[\rho]] \quad \equiv \quad \mathrm{cod}(d)[\overline{x}, \epsilon(pat)][\rho] \quad \equiv \quad \mathrm{cod}(d)[\overline{t}, u].$$

Finally, by anti-reduction $d(\overline{t}; w)$ is reducible at the adequate type.

The case of an eliminator over a universe follows the same pattern, the main modification being the organisation of the inductive hypothesis coming from the signature. For the case of a builder of negative type, the general case builds a reducibility proof now by coinduction. □

5.3 Consequences

Using the fundamental lemma and the definition of the logical relation on positive inert types, we obtain as a direct consequence that any term $\Sigma\,;\cdot \vdash t :^{\mathbb{ty}} K(\overline{u})$ of positive type is convertible to an inert constant introducing K or a neutral term. The following lemma ensures that there is no neutral term in $\mathbb{ty}$, so Theorem 5 and its immediate corollary Theorem 6 follow.

Lemma 15. *Closed neutrals belong to isolated sorts:*

$$\Sigma\,;\cdot \vdash n :^{s} A \wedge \mathrm{neutral}\, n \implies \mathrm{isolated}(s) \vee (A = \square^{s'} \wedge \mathrm{isolated}(s'))$$

Proof. The proof proceed by induction on the neutrality of n. The variable case is impossible since the context is empty, and the application case $n = n'\ t$ proceed by induction on n' using inversions on the typing derivation to show that n' is typed with a Π type in the empty context, at the same sort s as n, hence s is isolated. The important case $n = d(\overline{p}; w)$ consider two cases depending on the neutrality of w. If w is not neutral, $d \in \Sigma$ does not satisfy progress, so its codomain must satisfy the conclusion of the lemma. If w is neutral, it is again well-typed in an empty context by inversion on the typing derivation, so the type of w satisfies the conclusion of the lemma by induction hypothesis, and we conclude because the active constant $d \in \Sigma$ must preserve isolation. □

Decidability of conversion is proven by defining an algorithmic version of the conversion of two terms t and u which basically amounts to computing the whnf of t and u, compare their head, and apply the algorithm recursively if necessary. Correctness of algorithmic conversion is easy as the rules used are particular cases of typed conversion (Fig. 3). Then, it is shown that this algorithmic conversion is also complete by replaying Theorem 14 with a definition of the logical relation using algorithmic conversion instead of typed conversion. Actually, the formalization of [Abel et al. 2018] factorizes the two proofs of the fundamental lemma by defining an abstract interface to both algorithmic conversion and typed conversion and use this interface in the definition of the logical relation instead. Then, to get decidability of type checking, we can simply rely on the work of Lennon-Bertrand [2021] on bidirectional type-checking, which defines an algorithmic version of type-checking provided that the theory enjoys subject reduction and decidability of conversion.

6 EXTENSIONALITY

Our parametrization of MuTT in §2 only allows us to extend conversion through the introduction of new rewrite rules. However, some extensions of conversion such as extensionality principles are inherently undirected and cannot be specified with reduction rules, but must directly extend conversion. In this section, we do not provide a generic mechanism to enrich conversion with extensionality principles, a challenging goal that we leave for future work, but we remark that the logical relation naturally justifies them on two compelling examples.

Primitive projections. The definition of negative dependent sum in §3.1 (and more generally, any record type) can be equipped with the following extensionality principle:

$$\frac{\Sigma\,;\Gamma \vdash t :^{\mathbb{ty}} \Sigma\ A\ B}{\Sigma\,;\Gamma \vdash \mathtt{pair}(A, B, \mathtt{fst}(A, B, t), \mathtt{snd}(A, B, t)) \equiv t :^{\mathbb{ty}} \Sigma\ A\ B}$$

This conversion rule can be added to the system by postulating it when t is neutral. The logical relation framework then straightforwardly shows that the conversion is valid on any term, as any term of type $\Sigma\ A\ B$ reduces to a whnf which is either a `pair`, in which case the equality holds by computation of `fst` and `snd`; or it is a neutral term, in which case the equality holds with the new conversion rule. Then, it suffices to remark that algorithmic conversion can also be extended with this new conversion rule on neutral terms without compromising decidability.

Strict Propositions. Gilbert et al. [2019] propose the introduction of a new sort $\mathbb{sP}$ of *strict propositions* to Agda[2] and Coq[3]. The characteristic feature of $\mathbb{sP}$ is its definitional proof irrelevance, e.g. any two inhabitants p, q of a type P in $\mathbb{sP}$ are convertible:

$$\frac{\Sigma\,;\Gamma \vdash p :^{\mathbb{sP}} P \qquad \Sigma\,;\Gamma \vdash q :^{\mathbb{sP}} P}{\Sigma\,;\Gamma \vdash p \equiv q :^{\mathbb{sP}} P}$$

As explained by Gilbert et al. [2019], to encode such a sort of strict proposition in MuTT, it is enough to introduce a new sort $\mathbb{sP}$ with a single empty inductive type $\Sigma\,;\vdash \bot : \mathbb{sP}$ and its eliminator to $\mathbb{ty}$, together with a conversion rule equating any two neutral terms at this type.

7 RELATED WORK

Extending type theories. Pure Type Systems (PTS) [Barendregt et al. 2013] is a general framework for defining type theories based on λ-calculus extended with additional sort constants. Metatheoretical results such as consistency, subject reduction and normalisation have been established for classes of PTS and their extensions, for instance with cumulativity [Luo 1990] but the computational content is usually entirely defined from β-reduction. Allais et al. [2013] extend conversion with a fixed set of additional equations between neutral terms for a simply typed language, with type theory left as a future work goal. Their work use a similar methodology with a logical relation to show that conversion can be reduced to a standard reduction path. Basold and Geuvers [2016] propose a type theory with a generic scheme to define inductive and coinductive types uniformly. MuTT develops beyond their treatment in two orthogonal directions, supporting universes and allowing types that are not necessarily inductive or coinductive.

Rewriting in type theory. Our setting to define rewrite rules is based on the recent work of Cockx et al. [2021] but combining rewrite systems and type systems stems from the work of Tannen [1988], extending simply typed lambda-calculus with higher-order rewrite rules. This framework was later taken to dependent type theory by Barbanera et al. [1997]. They extend the Calculus of Constructions with first- and higher-order rewrite rules, provided the higher-order rules do not introduce any critical pairs. Walukiewicz-Chrzaszcz [2003] prove subject reduction for another variant of the Calculus of Constructions with a more general notion of higher-order rewrite rules and completeness and consistency of this system has been studied in [Walukiewicz-Chrzaszcz and Chrzaszcz 2006]. The Calculus of Algebraic Constructions [Blanqui 2005] is another extension of the Calculus of Constructions with a restricted form of higher-order rewrite rules. It also provides criteria for checking subject reduction and strong normalization. All those work serves as the base

[2] https://agda.readthedocs.io/en/v2.6.0/language/prop.html
[3] Since Coq 8.10: https://coq.inria.fr/doc/addendum/sprop.html

to our present work, and we do not claim any originality with respect to our termination criteria which is basically enforced by typing conditions in the definition of a well-formed signature (§2.3).

Modal type theories. Modalities have recently gained traction to extend type theory in a variety of directions [Birkedal et al. 2020; Kavvos 2019; Nuyts and Devriese 2018; Rijke et al. 2020; Schreiber and Shulman 2012; Shulman 2018], supporting the addition of new logical principles and constraints on the structure of type theoretical judgments. In order to accommodate the zoo of modalities required for different applications, general frameworks parametrized by a 2-category of modes have been proposed, first in a simply typed setting [Licata et al. 2017], and gradually being adapted to a dependent setting [Birkedal et al. 2020; Gratzer et al. 2020, 2019]. Modalities, in particular non-lex comonads that modify the action of substitutions, encompass a wider setting than what we present in this work at the cost of a more complex metatheory. To this day, beyond state-of-the-art experiments, no proof assistant implementation support parametrized modalities. Our approach assumes standard context management and substitution propagation so should be more readily compatible with existing mainstream proof assistant such as Coq or Agda for an implementation.

Logical relations, type theory and categorical models. Since Plotkin's seminal work [Plotkin 1973], logical relations have been used pervasively to prove metatheoretical properties of programming languages and type theory [Mitchell 1991]. A categorical perspective on these techniques have been developed over the last three decades [Fiore 2002; Mitchell and Scedrov 1992; Shulman 2015; Sterling and Harper 2020; Sterling and Spitters 2018], providing efficient but rarely effective methods to prove normalization. Abel et al. [2007] apply these techniques to dependent type type theory, while Coquand [2019, 2021] uses a so-called reduction-free variant of logical relations. Abel et al. [2018] provide the first mechanization of logical relations to prove decidability of type checking of type theory in itself, on which we build. Such mechanized developments remain to date a difficult task as witnessed by the recent POPLMark reloaded challenge [Abel et al. 2019].

8 CONCLUSION

We have presented a generic multiverse type theory MuTT in which multiple, possibly incompatible type universes can safely cohabit without endangering its meta-theoretical properties. This new sort system provides a type-theoretic mechanism to separate incompatible computational or logical features, which can further be used to mediate between universes, *e.g.*, using specific new constants that make bridges between universes. Beyond the simple instances that we present here, we expect that many models of MLTT [Altenkirch et al. 2019; Boulier et al. 2017; Pédrot and Tabareau 2020] have interesting presentations in MuTT capturing their computational behaviour. Parametrized extensions of conversion as presented in §6 is an important future milestone to that endeavour. A natural next step is to make the theory sort-polymorphic, so that sort-agnostic definitions can be shared more easily between universes, extending the existing universe-level polymorphism that is implemented in today's proof assistants [Sozeau and Tabareau 2014; The Agda Development Team 2021].

REFERENCES

Andreas Abel, Klaus Aehlig, and Peter Dybjer. 2007. Normalization by Evaluation for Martin-Löf Type Theory with One Universe. In *Proceedings of the 23rd Conference on the Mathematical Foundations of Programming Semantics, MFPS 2007, New Orleans, LA, USA, April 11-14, 2007 (Electronic Notes in Theoretical Computer Science, Vol. 173)*, Marcelo Fiore (Ed.). Elsevier, 17–39. https://doi.org/10.1016/j.entcs.2007.02.025

Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. 2019. POPLMark reloaded: Mechanizing proofs by logical relations. *J. Funct. Program.* 29 (2019), e19. https://doi.org/10.1017/S0956796819000170

Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: programming infinite structures by observations. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 27–38. https://doi.org/10.1145/2429069.2429075

Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2018. Decidability of conversion for type theory in type theory. *Proc. ACM Program. Lang.* 2, POPL (Jan. 2018), 23:1–23:29. https://doi.org/10.1145/3158111

Guillaume Allais, Conor McBride, and Pierre Boutillier. 2013. New equations for neutral terms: a sound and complete decision procedure, formalized, Stephanie Weirich (Ed.). ACM Press, Boston, Massachusetts, USA, 13–24. https://doi.org/10.1145/2502409.2502411

Thorsten Altenkirch, Simon Boulier, Ambrus Kaposi, and Nicolas Tabareau. 2019. Setoid Type Theory - A Syntactic Translation. In *Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11825)*, Graham Hutton (Ed.). Springer, 155–196. https://doi.org/10.1007/978-3-030-33636-3_7

Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. 2016. Extending Homotopy Type Theory with Strict Equality. *Computer Science Logic* (2016).

Franco Barbanera, Maribel Fernández, and Herman Geuvers. 1997. Modularity of Strong Normalization in the Algebraic-lambda-Cube. *Journal of Functional Programming* 7, 6 (1997), 613–660.

Hendrik Pieter Barendregt, Wil Dekkers, and Richard Statman. 2013. *Lambda Calculus with Types*. Cambridge University Press. http://www.cambridge.org/de/academic/subjects/mathematics/logic-categories-and-sets/lambda-calculus-types

Gilles Barthe and Tarmo Uustalu. 2002. CPS Translating Inductive and Coinductive Types. In *Proceedings of Partial Evaluation and Semantics-based Program Manipulation* (Portland, Oregon). ACM, 131–142.

Henning Basold and Herman Geuvers. 2016. Type Theory based on Dependent Inductive and Coinductive Types, Martin Grohe, Eric Koskinen, and Natarajan Shankar (Eds.). ACM Press, New York, NY, USA, 327–336. https://doi.org/10.1145/2933575.2934514

Lars Birkedal, Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. 2020. Modal dependent type theory and dependent right adjoints. *Math. Struct. Comput. Sci.* 30, 2 (2020), 118–138. https://doi.org/10.1017/S0960129519000197

Frédéric Blanqui. 2005. Definitions by rewriting in the Calculus of Constructions. *Mathematical Structures in Computer Science* 15, 1 (2005), 37–92. https://doi.org/10.1017/S0960129504004426

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

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

Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. 2021. The taming of the rew: a type theory with computational assumptions. *Proc. ACM Program. Lang.* 5, POPL (Jan. 2021), 1–29. https://doi.org/10.1145/3434341

Thierry Coquand. 2019. Canonicity and normalization for dependent type theory. *Theor. Comput. Sci.* 777 (2019), 184–191. https://doi.org/10.1016/j.tcs.2019.01.015

Thierry Coquand. 2021. Reduction Free Normalisation for a proof irrelevant type of propositions. *CoRR* abs/2103.04287 (2021). arXiv:2103.04287 https://arxiv.org/abs/2103.04287

Thierry Coquand and Gérard Huet. 1988. The Calculus of Constructions. *Information and Computation* 76, 2-3 (Feb. 1988), 95–120.

Marcelo Fiore. 2002. Semantic Analysis of Normalisation by Evaluation for Typed Lambda Calculus. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Pittsburgh, PA, USA) *(PPDP '02)*. Association for Computing Machinery, New York, NY, USA, 26–37. https://doi.org/10.1145/571157.571161

Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. 2019. Definitional proof-irrelevance without K. *Proc. ACM Program. Lang.* 3, POPL (Jan. 2019), 3:1–3:28. https://doi.org/10.1145/3290316

V. Glivenko. 1929. Sur Quelques Points de la Logique de M. Brouwer. *Bulletins de la classe des sciences* 15 (1929), 183–188.

Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. 2020. Multimodal Dependent Type Theory, See [Hermanns et al. 2020], 492–506. https://doi.org/10.1145/3373718.3394736

Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. 2019. Implementing a modal dependent type theory. See[ICFP 2019 2019], 107:1–107:29. https://doi.org/10.1145/3341711

Hugo Herbelin. 2005. On the Degeneracy of Sigma-Types in Presence of Computational Classical Logic. In *Seventh International Conference, TLCA '05, Nara, Japan. April 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3461)*, Pawel Urzyczyn (Ed.). Springer, 209–220.

Hugo Herbelin and Arnaud Spiwack. 2013. The Rooster and the Syntactic Bracket. In *19th International Conference on Types for Proofs and Programs, TYPES 2013, April 22-26, 2013, Toulouse, France (LIPIcs, Vol. 26)*, Ralph Matthes and Aleksy Schubert (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 169–187. https://doi.org/10.4230/LIPIcs.TYPES.2013.169

Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller (Eds.). 2020. *Proceedings of the 35th ACM/IEEE Symposium on Logic in Computer Science (LICS 2020)*. ACM, Saarbrücken, Germany. https://doi.org/10.1145/3373718

ICFP 2019 2019.

G. A. Kavvos. 2019. Modalities, cohesion, and information flow. *Proc. ACM Program. Lang.* 3, POPL (2019), 20:1–20:29. https://doi.org/10.1145/3290333

Meven Lennon-Bertrand. 2021. Complete Bidirectional Typing for the Calculus of Inductive Constructions. In *12th International Conference on Interactive Theorem Proving (ITP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ITP.2021.24

Pierre Letouzey. 2004. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. Ph.D. Dissertation. Université Paris-Sud.

Daniel R. Licata, Michael Shulman, and Mitchell Riley. 2017. A Fibrational Framework for Substructural and Modal Logics. In *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK (LIPIcs, Vol. 84)*, Dale Miller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 25:1–25:22. https://doi.org/10.4230/LIPIcs.FSCD.2017.25

Zhaohui Luo. 1990. *An Extended Calculus of Constructions*. Ph.D. Dissertation. Department of Computer Science, University of Edinburgh.

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

Per Martin-Löf. 1975. An intuitionistic theory of types: predicative part. In *Logic Colloquium '73, Proceedings of the Logic Colloquium*, H.E. Rose and J.C. Shepherdson (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 80. North-Holland, 73–118.

Per Martin-Löf. 2006. 100 years of Zermelo's axiom of choice: what was the problem with it? *Comput. J.* 49, 3 (2006), 345–350. https://doi.org/10.1093/comjnl/bxh162

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

John C. Mitchell. 1991. *Type Systems for Programming Languages*. MIT Press, Cambridge, MA, USA, 365–458.

John C. Mitchell and Andre Scedrov. 1992. Notes on Sconing and Relators. In *Computer Science Logic, 6th Workshop, CSL '92, San Miniato, Italy, September 28 - October 2, 1992, Selected Papers (Lecture Notes in Computer Science, Vol. 702)*, Egon Börger, Gerhard Jäger, Hans Kleine Büning, Simone Martini, and Michael M. Richter (Eds.). Springer, 352–378. https://doi.org/10.1007/3-540-56992-8_21

Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (July 1991), 55–92.

Ulf Norell. 2009. Dependently Typed Programming in Agda. In *Advanced Functional Programming (AFP 2008) (Lecture Notes in Computer Science, Vol. 5832)*. Springer-Verlag, 230–266.

Andreas Nuyts and Dominique Devriese. 2018. Degrees of Relatedness: A Unified Framework for Parametricity, Irrelevance, Ad Hoc Polymorphism, Intersections, Unions and Algebra in Dependent Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 779–788. https://doi.org/10.1145/3209108.3209119

David Parnas. 1972. On the criteria for decomposing systems into modules. *Commun. ACM* 15, 12 (Dec. 1972), 1053–1058.

Christine Paulin-Mohring. 1993. Inductive Definitions in the System Coq - Rules and Properties. In *Typed Lambda Calculi and Applications*, Marc Bezem and Jan Friso Groote (Eds.). https://doi.org/10.1007/BFb0037116

Christine Paulin-Mohring. 2015. Introduction to the Calculus of Inductive Constructions. In *All About Proofs, Proofs for All*, Bruno Woltzenlogel Paleo and David Delahaye (Eds.). College Publications.

Pierre-Marie Pédrot and Nicolas Tabareau. 2020. The fire triangle: how to mix substitution, dependent elimination, and effects. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 58:1–58:28.

Pierre-Marie Pédrot, Nicolas Tabareau, Hans Jacob Fehrmann, and Éric Tanter. 2019. A Reasonably Exceptional Type Theory. 3, ICFP, Article 108 (July 2019), 29 pages. https://doi.org/10.1145/3341712

Gordon D. Plotkin. 1973. Lambda-definability and logical relations. https://www.cl.cam.ac.uk/~nk480/plotkin-logical-relations.pdf

Pierre-Marie Pédrot and Nicolas Tabareau. 2018. Failure is Not an Option - An Exceptional Type Theory. In *Proceedings of the 27th European Symposium on Programming Languages and Systems (ESOP 2018) (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer-Verlag, Thessaloniki, Greece, 245–271.

Egbert Rijke, Michael Shulman, and Bas Spitters. 2020. Modalities in homotopy type theory. *Log. Methods Comput. Sci.* 16, 1 (2020). https://doi.org/10.23638/LMCS-16(1:2)2020

Urs Schreiber and Michael Shulman. 2012. Quantum Gauge Field Theory in Cohesive Homotopy Type Theory. In *Proceedings 9th Workshop on Quantum Physics and Logic, QPL 2012, Brussels, Belgium, 10-12 October 2012 (EPTCS, Vol. 158)*, Ross Duncan and Prakash Panangaden (Eds.). 109–126. https://doi.org/10.4204/EPTCS.158.8

Michael Shulman. 2015. Univalence for inverse diagrams and homotopy canonicity. *Math. Struct. Comput. Sci.* 25, 5 (2015), 1203–1277. https://doi.org/10.1017/S0960129514000565

Michael Shulman. 2018. Brouwer's fixed-point theorem in real-cohesive homotopy type theory. *Math. Struct. Comput. Sci.* 28, 6 (2018), 856–941. https://doi.org/10.1017/S0960129517000147

Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 2020. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 8:1–8:28. https://doi.org/10.1145/3371076

Matthieu Sozeau and Nicolas Tabareau. 2014. Universe Polymorphism in Coq. In *Interactive Theorem Proving*, Gerwin Klein and Ruben Gamboa (Eds.). Springer International Publishing, Cham, 499–514.

Jonathan Sterling. 2019. Algebraic Type Theory and Universe Hierarchies. arXiv:1902.08848 http://arxiv.org/abs/1902.08848

Jonathan Sterling and Robert Harper. 2020. Logical Relations as Types: Proof-Relevant Parametricity for Program Modules. *CoRR* abs/2010.08599 (2020). arXiv:2010.08599 https://arxiv.org/abs/2010.08599

Jonathan Sterling and Bas Spitters. 2018. Normalization by gluing for free λ-theories. *CoRR* abs/1809.08646 (2018). arXiv:1809.08646 http://arxiv.org/abs/1809.08646

Nikhil Swamy, Juan Chen, and Ben Livshits. 2013. Verifying Higher-order Programs with the Dijkstra Monad. In *ACM Programming Language Design and Implementation (PLDI) 2013*.

Val Tannen. 1988. Combining Algebra and Higher-Order Types. In *Proceedings, Third Annual Symposium on Logic in Computer Science, 5-8 July 1988, Edinburgh, Scotland, UK*. IEEE Computer Society, 82–90. https://doi.org/10.1109/LICS.1988.5103

The Agda Development Team. 2021. Universe Levels. https://agda.readthedocs.io/en/v2.6.2/language/universe-levels.html

The Coq Development Team. 2020. *The Coq proof assistant reference manual.* https://coq.inria.fr/refman/ Version 8.12.

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics.* http://homotopytypetheory.org/book, Institute for Advanced Study.

Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. See[ICFP 2019 2019].

Vladimir Voevodsky. 2013. *A simple type system with two identity types.* Unpublished notes, http://uf-ias-2012.wikispaces.com/file/view/HTS.pdf.

Daria Walukiewicz-Chrzaszcz. 2003. Termination of rewriting in the Calculus of Constructions. *Journal of Functional Programming* 13, 2 (2003), 339–414. https://doi.org/10.1017/S0956796802004641

Daria Walukiewicz-Chrzaszcz and Jacek Chrzaszcz. 2006. Consistency and Completeness of Rewriting in the Calculus of Constructions. In *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4130)*, Ulrich Furbach and Natarajan Shankar (Eds.). Springer, 619–631. https://doi.org/10.1007/11814771_50

Alfred North Whitehead and Bertrand Russell. 1910. *Principia Mathematica.* Cambridge University Press, Cambridge.