There is Life in the Universes Beyond ω

ANONYMOUS AUTHOR(S)

The first draft of Martin-Löf's type theory proposed the assumption Type:Type. Subsequently, universe levels have been introduced to avoid the resulting inconsistencies by assuming Type_i:Type_{i+1}. Proof assistants based on type theory support such universe levels to varying degree, but they impose restrictions that can make coding awkward.

Specifically, we consider the ramifications of Agda's approach to handling levels using a denotational semantics of a stratified version of System F as a motivating example. We propose a simple fix that extends Agda's capabilities for handling universe levels parametrically up to ε_0.

Additional Key Words and Phrases: Dependent types, universes, ordinal numbers

ACM Reference Format:
Anonymous Author(s). 2018. There is Life in the Universes Beyond ω. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX).* ACM, New York, NY, USA, 16 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The origin of universe levels.

How do universe levels work in Agda?

What becomes awkward with Agda's approach?

How do we propose to fix it?

Contributions.

2 Preliminaries

Agda, Ordinals, IR-Universes

2.1 Universes in Agda

TODO: typeset in Agda style

Agda contains significant support for an infinite hierarchy of universes [10]. It provides an abstract datatype Level of universe levels along with constants zero, suc, and _ ⊔ _ that denote the base-level of the hierarchy, the successor, and the maximum of two levels. The level-parametric type Set i stands for the universe at level i and it obeys the typing Set i : $ASeti$suc i. Base types inhabit the universe Set zero. Type formation handles universe levels in the same way as finitely stratified System F [7]. That is,

- if A_1 : Set i_1 and A_2 : Set i_2, then $A_1 \longrightarrow A_2$: Set $(i_1 \sqcup i_2)$;
- if α : Set i is a type variable and A : Set j, then $\forall\alpha.A$: Set $(suc\ i \sqcup j)$.

To avoid inconsistencies, Agda does not allow pattern matching on the type Level. However, quantification over Level is allowed and results in a type at level ω (if the level-typed variable is used in a significant way). Unfortunately, levels ω and higher are **not** handled in a parametric way in Agda. Rather there are predefined types Set_ω, $\mathsf{Set}_{\omega 1}$, $\mathsf{Set}_{\omega 2}$, and so on, for the levels ω, $\omega + 1$, $\omega + 2$. This design has some limitations, as most notions in the standard library (prominently the equality type and its supporting infrastructure, but also parameteric types like sums, pairs, lists, and so on) are defined in a level-parametric way, and thus they cannot be used with Set_ω and higher universe levels.

For convenience, Agda's unification algorithm has special treatment for the Level datatype and its operators [10]:

- Idempotence: $a \sqcup a \equiv a$.
- Associativity: $(a \sqcup b) \sqcup c$ is the same as $a \sqcup (b \sqcup c)$.
- Commutativity: $a \sqcup b$ is the same as $b \sqcup a$.
- Distributivity of suc over $\sqcup$: $\mathsf{suc}(a \sqcup b)$ is the same as $\mathsf{suc}\ a \sqcup \mathsf{suc}\ b$.
- Neutrality of zero: $a \sqcup \mathsf{zero}$ is the same as a.
- Subsumption: $a \sqcup \mathsf{suc}\ a$ is the same as $\mathsf{suc}\ a$.

In other words, the level structure is a join-semilattice with an inflationary endomorphism [1].

TODO: Cumulativity

2.2 Ordinals

Ordinal numbers are an important concept in mathematics and computer science. They are closely related to well-ordered sets as any well-ordered set is order-isomorphic to an ordinal. The significance to computer science is that such well-orders can be used for termination proofs.

The best known construction of ordinals is a set-theoretic one due to von Neumann. It starts with the smallest ordinal 0 represented by the empty set $\emptyset$. To construct the successor of an ordinal α, we define $\alpha + 1 := \alpha \cup \{\alpha\}$. This way, we construct $1, 2, 3, \ldots$. Then, we can scoop them all up into the smallest limit ordinal $\omega = \{0, 1, 2, 3, \ldots\}$, which contains 0 and all finite applications of the successor to it. We continue with $\omega + 1 = \omega \cup \{\omega\}$ and carry on until we build the next limit ordinal $\omega \cdot 2$, then $\omega \cdot 3$, and so on. Constructing the limit at this level yields ω^ω and continuing further leads to ω^{ω^ω}, $\omega^{\omega^{\omega^\omega}}$, and so on. The limit of this sequence is ε_0 which is the smallest ordinal that fulfills the equation $\varepsilon_0 = \omega^{\varepsilon_0}$.

In the context of this paper, we plan to use ordinals as universe levels where the well-ordering avoids collapsing levels. The construction of ordinals does not stop at ε_0, but we wish to carve out a particular set of ordinals that fits well in an implementation context. Concretely, we consider ordinals less than ε_0 as they can be represented by binary trees. To see this, recall that every ordinal α can be written in Cantor normal form

$$\alpha = \omega^{\beta_1} + \omega^{\beta_2} + \cdots + \omega^{\beta_n}$$

for some $n \geq 0$ and ordinals β_i such that $\beta_1 \geq \beta_2 \geq \cdots \geq \beta_n$. If $\alpha < \varepsilon_0$, then it can be shown that each exponent satisfies $\beta_i < \alpha$. If we, again, write $\beta_i = \omega^{\gamma_1} + \cdots + \omega^{\gamma_m}$ in Cantor normal form, then clearly $\gamma_j < \beta_i < \alpha$. As the ordering on ordinals is a well-order, we know this decreasing sequence must terminate and we obtain a finite representation for each ordinal less than ε_0.

Forsberg et al. [4] developed Agda formalizations for three equivalent representations for precisely this set of ordinals. They define addition and multiplication of ordinals, prove the principle of transfinite induction, and use that to prove that the represented subset of ordinals is well-ordered.

We build on one of their representations, called MutualOrd, define some additional operations, and prove some properties which are needed in the context of universe levels.[1] In particular, we prove the laws of a join-semilattice with an inflationary endomorphism.

TODO: Insert

- Definition of MutualOrd
- Explanation
- Example(s)
- any additional properties that we had to prove

2.3 Encoding Universes

While Agda has built-in support for universes, it is possible to "compress" certain universe hierarchies into Set. Kovács [6] constructs a very general model of cumulative universe hierarchies, which situated entirely in Set 0 and Set 1. The construction requires induction-recursion [3], so that the model can be constructed in Agda along with its consistency proof.

Kovács's implementation in Agda is along the lines of McBride [8]'s universe construction. Here we just recall the most important definitions of his construction of TTDL (type theory with dependent levels).

A *level structure* provides a set Lvl of levels, a strict linear order $<$, evidence that this order is propositional, and a composition operator that embodies transitivity of the order. Universes can be indexed by any level structure, for example, natural numbers for ordinals less than ω, pairs of natural numbers for ordinals less than $\omega + \omega$, and in fact any type-theoretic ordinal.

```
Lvl    : Set
_<_    : Lvl → Lvl → Set
<-prop : ∀ {i j} {p q : i < j} → p ≡ q
_∘_    : ∀ {i j k} → j < k → i < j → i < k
```

The inductive type U^{ir} of universe codes is defined mutually recursively with its interpretation El^{ir} in Set.

```
data U^ir {i}(l : ∀ (j : Lvl) → j < i → Set) : Set      El^ir : ∀ {i l} → U^ir {i} l → Set

  U'   : ∀ {j} → j < i → U^ir l                          El^ir {_}{l}(U' p) = l _ p
  ℕ'   : U^ir l                                          El^ir ℕ'       = ℕ
  ⊤'   : U^ir l                                          El^ir ⊤'       = ⊤
  Π'   : (a : U^ir l) → (El^ir a → U^ir l) → U^ir l      El^ir (Π' a b) = ∀ x → El^ir (b x)
  Lvl' : U^ir l                                          El^ir Lvl'     = Lvl
  _<'_ : Lvl → Lvl → U^ir l                              El^ir (i <' j) = i < j
```

The actual universes are defined by induction on the accessibility of levels. Accessibility is formalized in the standard way by the type Acc with single constructor acc.

```
U< : ∀ {i} {{_ : Acc _<_ i}} j → j < i → Set
U< {i} {{acc f}} j p = U^ir {j} (U< {j}{{f p}})
```

[1]Technically, the mechanization of Forsberg et al. [4] relies on cubical Agda [?]. We back-ported the definitions for MutualOrd to standard Agda, as cubical is not needed for this representation.

With that we define the semantic hierarchy and its decoding function.

```
U : Lvl → Set - semantic universe               El : ∀ {i} → U i → Set
U i = Uⁱʳ {i} (U< {i} {{wf}})                    El {i} = Elⁱʳ {i}{U< {i}{{wf}}}
```

To fit in with the way that stratified System F (as well as Agda) handles levels, we need a non-strict lift operator on universes and proof that lifting does not change the semantic representation.

```
Lift≤ : ∀ {i j} → i ≤ j → U i → U j             ElLift≤ : ∀ {i j} p a → El (Lift≤ {i}{j} p a) ≡ El a
```

With these operators, Kovács defines an alternative Π-type encoding that fits perfectly with the requirements of stratified System F, namely that the level of a function type is the maximum of the levels of argument and return types.

```
Π” : ∀ {i j}(a : U i) → (El a → U j) → U (i ⊔ j)
Π” {i}{j} a b = Π’ (Lift≤ (⊔₁ i j) a) λ x → Lift≤ (⊔₂ i j) (b (coe (ElLift≤ (⊔₁ i j) a) x))
```

Here, $\sqcup_1$ and $\sqcup_2$ are proofs that the first (second) argument of $\sqcup$ is less than or equal to their maximum. The function coe transforms a type by applying an equality to it.

There is also a definition of a non-dependent function code.

```
_⇒’_ : ∀ {i l} → Uⁱʳ {i} l → Uⁱʳ l → Uⁱʳ l
a ⇒’ b = Π’ a λ _ → b
```

Kovács [6] also provides a worked example of a level structure for ordinals up to $\omega + \omega$, corresponding to the current version of Agda, as well as some infrastructure for using type-theoretic ordinals as levels.

3 Running Example

As a running example for demonstrating various encodings, we consider different extensions of finitely stratified System F [7]. More precisely, we start from an intrinsically typed encoding of types and expressions, then we construct denotational semantics for different encodings and discuss their respective merits.

We choose this system as it is significant, presents non-trivial challenges, and it has been studied in the literature. Our encoding of syntax is inspired by Chapman et al. [2], who develop the syntactic metatheory of System-Fω (without stratification). It has been picked up by Saffrich et al. [9], who give denotational and operational semantics for finitely stratified System-F and develop a logical relation for it. Hubers and Morris [5] use a similar syntax representation for a finitely stratified version of System-Fω extended with qualified types. They also develop a denotational semantics for their calculus. All these papers come with Agda formalizations.

The following subsections gives some excerpts from our Agda formalization which is available as supplemental material. We generally omit definitions for renamings and substitutions in the paper as they closely follow the abovementioned references. While the definitions are available in the supplemental material, some scaffolding lemmas (e.g., about composition of substitutions) are only postulated as their proofs can either be found in the abovementioned references or their proofs are very similar to proofs found there.

3.1 Types

We start with an intrinsically-typed encoding of types and expressions of finitely stratified System F. Like Saffrich et al. [9] and Hubers and Morris [5], we rely on Agda’s Level type to represent levels syntactically. This choice has the advantage that Agda's special treatment applies to syntactic levels (see 2.1). The representation of a type variable context is TEnv = List Level ranged over by Δ.

Types T take a type variable context Δ as a parameter and a level index. The variables ℓ, ℓ_1, … range over Level. The construction is standard. Base types live at level zero; function types live at the maximum level of argument and return types; type variables are represented by their position in the type variable context, i.e., as de Bruijn indices; and universal quantification introduces a new type variable at level ℓ.

```
data Type (Δ : TEnv) : Level → Set where
  Nat   : Type Δ zero
  _⇒_ : Type Δ ℓ₁ → Type Δ ℓ₂ → Type Δ (ℓ₁ ⊔ ℓ₂)
  `_    : ℓ ∈ Δ → Type Δ ℓ
  ∀α    : Type (ℓ :: Δ) ℓ′ → Type Δ (suc ℓ ⊔ ℓ′)
```

Variable contexts Γ are indexed by a type variable context. They are either empty, bind a variable to a type, or bind a type variable to a level. Correspondingly, variable access _ ∋ _ has two ways to skip over a binding. Accessing a variable across a type variable binding (in tskip) requires weakening the type found before the type binding.[2]

```
data EEnv : (Δ : TEnv) → Set where
  []    : EEnv Δ
  _::_  : Type Δ ℓ → EEnv Δ → EEnv Δ
  _::ℓ_ : (ℓ : Level) → EEnv Δ → EEnv (ℓ :: Δ)
```

```
data _∋_ : EEnv Δ → Type Δ ℓ → Set where
  here  : (T :: Γ) ∋ T
  there : Γ ∋ T → (T′ :: Γ) ∋ T
  tskip : Γ ∋ T → (ℓ ::ℓ Γ) ∋ Twk T
```

3.2 Expressions

Expressions e take a variable context Γ as a parameter and are indexed by a type, as usual. The first five cases are standard as for simply-typed lambda calculus. The type-lambda takes the level of the new type variable and a body expression that is typed in a level-extended variable context to construct an expression of type $\forall\alpha T$. Type application takes such an expression and a type of suitable level to construct an expression of type $T[T']$, that is, substituting the first variable in T by T'.

```
data Expr (Γ : EEnv Δ) : Type Δ ℓ → Set where
  #_     : ℕ → Expr Γ Nat
  `suc   : Expr Γ Nat → Expr Γ Nat
  `_     : Γ ∋ T → Expr Γ T
  λx_    : Expr (T₁ :: Γ) T₂ → Expr Γ (T₁ ⇒ T₂)
  _·_    : Expr Γ (T₁ ⇒ T₂) → Expr Γ T₁ → Expr Γ T₂
  Λ_⇒_ : (ℓ : Level) {T : Type (ℓ :: Δ) ℓ′} → Expr (ℓ ::ℓ Γ) T → Expr Γ (∀α T)
  _•_    : Expr Γ (∀α T) → (T′ : Type Δ ℓ) → Expr Γ (T [ T′ ]T)
```

4 Native Agda Semantics

The denotational semantics we consider here consists of two parts: the semantics of types and the semantics of expressions. Both semantics are compositional.

[2]Twk is a standard renaming on types that performs weakening.

This section explores mappings of types and expressions to corresponding Agda domains that are as direct as possible, hence the name "native".

4.1 Types

The native semantics maps a type at level ℓ to Set ℓ. Due to the presence of type variables, the semantics needs a type variable environment that maps a type variable to its denotation. There are several options in defining this environment.

(1) We may define it as a function as follows:

```
⟦_⟧Δ   : (Δ : TEnv) → Setω
⟦ Δ ⟧Δ = ∀ ℓ → ℓ ∈ Δ → Set ℓ
```

This definition lives in Agda's (fixed) Setω, so that equational reasoning at the type becomes inconvenient.

(2) We may also define it as an inductive datatype

```
data ⟦_⟧Δ : (Δ : TEnv) → Setω where
  []η   : ⟦ [] ⟧Δ
  _::η_ : Set ℓ → ⟦ Δ ⟧Δ → ⟦ ℓ :: Δ ⟧Δ
```

Again, this definition lives in Setω, although we might think of defining a function that computes the level of the defined type from the levels of the type variable context:

```
suc⊔Δ_ : TEnv → Level
suc⊔Δ [] = zero
suc⊔Δ (ℓ :: ℓs) = suc ℓ ⊔ suc⊔Δ ℓs
```

However, we cannot use this function as an argument to Set _ because Agda does not let us to define an inductive datatype where the level depends on an index type.

(3) Finally, we define the environment by induction:

```
⟦_⟧Δ        : (Δ : TEnv) → Set (suc⊔Δ Δ)
⟦ [] ⟧Δ     = ⊤
⟦ ℓ :: Δ ⟧Δ = Set ℓ × ⟦ Δ ⟧Δ
```

We adopt the last alternative to define the type semantics: A type at level ℓ is interpreted as an element of Set ℓ.

```
⟦_⟧T : (T : Type Δ ℓ) → ⟦ Δ ⟧Δ → Set ℓ
⟦ Nat   ⟧T η       = ℕ
⟦ ` x   ⟧T η       = lookup-η η x
⟦ T₁ ⇒ T₂ ⟧T η = ⟦ T₁ ⟧T η → ⟦ T₂ ⟧T η
⟦ ∀α T ⟧T η        = ∀ A → ⟦ T ⟧T (A ::η η)
```

Remark 1. *The key to make this semantics readable lies in our choice of the Level datatype for syntactic levels. If we chose, e.g., natural numbers for syntactic levels, the definitions for function types and universal types would become cluttered with transport lemmas caused by the injection function toLevel : Nat → Level. In the case of function types, the proof obligation would be toLevel l_1 ⊔ toLevel l_2 ≡ toLevel($l_1 \sqcup_{\mathbb{N}} l_2$). As demonstrated in Section 5, transport lemmas in the semantics of types are to be avoided because they proliferate into the semantics of expressions.*

4.2 Expressions

For the expression semantics, we also need a variable environment. We define it analogously to the type variable environment, with a function to compute the maximum level in the environment and another to build the environment structure. The latter is not surprising: nested pairs of values.

```
⨆Γ : EEnv Δ → Level                  ⟦_⟧Γ : (Γ : EEnv Δ) → ⟦ Δ ⟧Δ → Set (⨆Γ Γ)
⨆Γ []          = zero                ⟦ []      ⟧Γ η  = ⊤
⨆Γ (T :: Γ)   = lvlOf T ⊔ ⨆Γ Γ       ⟦ T :: Γ ⟧Γ η  = ⟦ T ⟧T η × ⟦ Γ ⟧Γ η
⨆Γ (ℓ ::ℓ Γ) = ⨆Γ Γ                  ⟦ ℓ ::ℓ Γ ⟧Γ η = ⟦ Γ ⟧Γ (drop-η η)
```

The first five cases of the expression semantics correspond again to the analogous construction for the simply-typed lambda calculus. The interpretation of a type-lambda at level ℓ is a function that takes an element A : Set ℓ and stores it in the type variable environment. The interpretation of a type application is essentially the interpretation of its underlying expression e, but the type has to be adjusted according to the type substitution. The type of the required equality eq shows that we trade the type interpretation of T' in the type variable environment for the substitution of T' into T. The lemmas involved are straightforward substitution lemmas.

```
⟦_⟧E : {Γ : EEnv Δ} (e : Expr Γ T) (η : ⟦ Δ ⟧Δ) (γ : ⟦ Γ ⟧Γ η) → ⟦ T ⟧T η
⟦ # n      ⟧E η γ = n
⟦ ‘suc e ⟧E η γ = sucℕ (⟦ e ⟧E η γ)
⟦ ‘ x      ⟧E η γ = lookup-γ γ x
⟦_⟧E {T = (T₁ ⇒ T₂)} {Γ} (λx e) η γ = λ x → ⟦ e ⟧E η (_::γ_ {T = T₁} {Γ = Γ} x γ)
⟦ e₁ · e₂ ⟧E η γ = ⟦ e₁ ⟧E η γ (⟦ e₂ ⟧E η γ)
⟦ Λ ℓ ⇒ e ⟧E η γ = λ (A : Set ℓ) → ⟦ e ⟧E (A ::η η) γ
⟦ _•_ {T = T} e T′ ⟧E η γ = coe eq (⟦ e ⟧E η γ (⟦ T′ ⟧T η))
  where eq : ⟦ T ⟧T (⟦ T′ ⟧T η ::η η) ≡ ⟦ T [ T′ ]T ⟧T η
        eq = trans (cong (λ η′ → ⟦ T ⟧T (⟦ T′ ⟧T η ::η η′)) (sym (⟦Tidₛ⟧σ _)))
                   (sym (⟦Tsub⟧T η _ T))
```

4.3 Discussion

To wrap up, the construction of a native Agda semantics for stratified System F is not too hard. One crucial point is choosing a good definition for the type variable environment. Because we chose a definition that does *not* require Setω, all equalities (in particular eq in the interpretation of type application) live in a suitable level-indexed universe and can be handled homogeneously by library definitions.

The transport equality eq in the interpretation of expressions appears to be unavoidable. So is another equality (not shown) in the definition of lookup-γ.

A final crucial point: The interpretation of types requires no transports because types are indexed by Agda's Level type, which is treated specially by Agda's normalization and unification procedures.

5 Level Quantification

As a new twist, we add level quantification (as in Agda) to the source language. To this end, we need a new syntactic category: level expressions. They take a level variable context LEnv = List ⊤ (ranged over by δ) as a parameter and are indexed by a Mode, ranged over by μ. The former determines the level variables that may appear and the latter determines whether a level expression denotes a finite level, less than ω, (fin) or if it is unrestricted (any).

```
data Lvl (δ : LEnv) : Mode → Set where
  `zero : Lvl δ fin
  `suc  : Lvl δ μ → Lvl δ μ
  `_    : tt ∈ δ → Lvl δ fin
  _`⊔_ : Lvl δ μ → Lvl δ μ → Lvl δ μ
  ⟨_⟩   : Lvl δ fin → Lvl δ any
  `ω    : Lvl δ any
```

From the indices we can see that level variables range over finite levels, the successor and maximum operators can be used with finite and unrestricted levels, and any expression containing ω is not finite. From the structure of level expressions, we can see that all denoted levels must be smaller than $\omega + \omega$.

Consequently, there are two interpretation functions for level expressions, one for finite expressions and a general one. We map finite expressions to natural numbers and general expressions to pairs of natural numbers (i, j) representing the ordinal $\omega \cdot i + j$. Both are relative to an interpretation ⟦_⟧δ of the level variables as finite levels, that is, natural numbers. Elements of such a level environment are essentially vectors of natural numbers.

```
⟦_⟧δ        : LEnv → Set
⟦ [] ⟧δ      = ⊤
⟦ _ :: δ ⟧δ = ℕ × ⟦ δ ⟧δ
```

```
  ⟦_⟧L′ : Lvl δ fin → ⟦ δ ⟧δ → ℕ                    ⟦_⟧L : Lvl δ any → ⟦ δ ⟧δ → OrdLvl
  ⟦ `zero  ⟧L′ κ    = ℕ.zero                         ⟦ ⟨ l ⟩    ⟧L κ = ℕ.zero , ⟦ l ⟧L′ κ
  ⟦ `suc l ⟧L′ κ    = ℕ.suc (⟦ l ⟧L′ κ)              ⟦ `suc l   ⟧L κ = sucL (⟦ l ⟧L κ)
  ⟦ l₁ `⊔ l₂ ⟧L′ κ = ⟦ l₁ ⟧L′ κ ⊔ℕ ⟦ l₂ ⟧L′ κ       ⟦ l₁ `⊔ l₂ ⟧L κ = ⟦ l₁ ⟧L κ ⊔ ⟦ l₂ ⟧L κ
  ⟦ ` x    ⟧L′ κ    = lookup-κ κ x                   ⟦ `ω       ⟧L κ = ω
```

5.1 Types

The definition of a type variable context and indexing into it is analogous to the construction of *expression* variable contexts and indexing in Section 3.2. We use lskip to skip over level bindings and Lwk for weakening of level expressions.

```
  data TEnv : LEnv → Set where                      data _∋_ : TEnv δ → Lvl δ any → Set where
    []   : TEnv δ                                     here  : (l :: Δ) ∋ l
    _::_ : Lvl δ any → TEnv δ → TEnv δ                there : Δ ∋ l → (l′ :: Δ) ∋ l
    ::l_ : TEnv δ → TEnv (tt :: δ)                    lskip  : Δ ∋ l → (::l Δ) ∋ Lwk l
```

The syntax of types is also very similar to the one from Section 3.1, with the following differences:

(1) The type variable context is indexed by level context δ.

(2) The index of type Level is replaced by an unrestricted level expression. It is unrestricted because the type of an expression can have level ω or higher due to the presence of level quantification.
(3) Level quantification $\forall\ell$ pushes a new level variable on the level context. It conservatively assumes that the new variable is used in a non-trivial way in the body of this abstraction. Hence, the level of this type is at least ω.

This structure models the provisions of Agda: There is level quantification over finite levels and this quantification may push levels to ω and higher, but all levels stay below $\omega + \omega$.

```
data Type (Δ : TEnv δ) : Lvl δ any → Set where
  Nat   : Type Δ ⟨ ‘zero ⟩
  ‘_    : Δ ∋ l → Type Δ l
  _⇒_ : Type Δ l₁ → Type Δ l₂ → Type Δ (l₁ ‘⊔ l₂)
  ∀α    : Type (l :: Δ) l′ → Type Δ (‘suc l ‘⊔ l′)
  ∀ℓ    : Type (::l Δ) (Lwk l) → Type Δ (‘ω ‘⊔ l)
```

Starting with the semantics of a type variable context, we draw from the inductive-recursive universe construction. Our interpretation of types takes two steps. The first encoding step interprets types as codes; the second step interprets these codes. Hence, a type variable context is interpreted as a vector of codes where a type variable at level l is mapped to a code for universe level l. (Recall that l is a level expression that has to be interpreted first.)

```
⟦_⟧Δ : (Δ : TEnv δ) → (κ : ⟦ δ ⟧δ) → Set
⟦ []      ⟧Δ κ   = ⊤
⟦ l :: Δ ⟧Δ κ = U (⟦ l ⟧L κ) × ⟦ Δ ⟧Δ κ
⟦ ::l Δ ⟧Δ κ   = ⟦ Δ ⟧Δ (drop-κ κ)
```

The encoding of the Nat type is obvious. Type variables are looked up in their environment. Function types are mapped to a non-dependent function code after lifting argument and result codes to their maximum level.

The encoding of universal quantification $\forall\alpha : l.T$ (with T at level l') involves a Π-type. The argument of this Π’-code is a type at some level l, so we have to give a universe code U’ with an argument pu that selects the correct level with a proof that it is strictly before the current one in the ordering. The current level is prescribed by type formation: (suc l) $\sqcup\, l'$. Now we reason $l <$ suc $l \leq$ (suc l) $\sqcup\, l'$, which is precisely the type of pu.

The second argument of Π’ takes some El^{ir} (U’ pu), which—by our construction—happens to be a universe of level l, so we can extend the environment η with it. However, we can only do so after fixing the accessibility proof in its type with the transport equality eq. We finish by constructing the encoding of type T in the extended environment and lift the resulting code at level l' to (suc l) $\sqcup\, l'$.

For the encoding of level quantification, we have to remember that quantification ranges over finite levels, i.e., natural numbers. Hence the code is a function Π’ with first argument ℕ’. The second argument takes some El^{ir} ℕ’, that is a natural number ℓ. It remains to encode T in the extended level variable environment, eliminate the weakening from T’s level with a transport equality and lift the resulting code to a level $\geq \omega$ to compensate for the level quantification.

```
encode : {Δ : TEnv δ} → (T : Type Δ l) → (κ : ⟦ δ ⟧δ) → (η : ⟦ Δ ⟧Δ κ) → U (⟦ l ⟧L κ)
encode Nat κ η          = ℕ’
encode (‘ x) κ η        = lookup-η η x
encode (T₁ ⇒ T₂) κ η = Lift≤ (⊔₁ _ _) (encode T₁ κ η) ⇒’ Lift≤ (⊔₂ _ _) (encode T₂ κ η)
encode {Δ = Δ} (∀α {l = l} T) κ η =
  let pu = <≤-trans ℕ*ℕ.<suc (⊔₁ _ _) in
  Π' (U' pu) λ A →
  let eq = U^ir & ext (λ j → ext (λ p → (λ acc → (U< {⟦ l ⟧L κ} {| acc |} j p)) & (Acc-prop _ wf))) in
  let S = encode T κ (_::η_ {l = l} {Δ = Δ} {κ = κ} (coe eq A) η) in
  Lift≤ (⊔₂ _ _) S
encode (∀ℓ {l = l} T) κ η = Π' ℕ' λ ℓ →
  let S = coe (U & ⟦Lwk⟧L l κ ℓ) (encode T (ℓ ::κ κ) η) in
  Lift≤ (⊔₂ ω _) S
```

The semantics of a type is then a set obtained by interpretation of the code.

```
⟦_⟧T : {Δ : TEnv δ} (T : Type Δ l) (κ : ⟦ δ ⟧δ) (η : ⟦ Δ ⟧Δ κ) → Set
⟦ T ⟧T κ η = El (encode T κ η)
```

5.2 Expressions

For expressions, we follow the same schema as before in Section 3.2. However, the expression context (ranged over by Γ) now registers *three* different kinds of bindings: types of variables, levels of type variables, and level variables. Correspondingly, the indexing machinery for variables also has additional constructors tskip and lskip to skip over type variable and level variable bindings.

```
data EEnv : (Δ : TEnv δ) → Set where
  []    : EEnv Δ
  _::_  : Type Δ l → EEnv Δ → EEnv Δ
  _::l_ : (l : Lvl δ any) → EEnv Δ → EEnv (l :: Δ)
  ::l_  : EEnv Δ → EEnv (::l Δ)
```

```
data _∋_ : EEnv Δ → Type Δ l → Set where
  here  : (T :: Γ) ∋ T
  there : Γ ∋ T → (T' :: Γ) ∋ T
  tskip : Γ ∋ T → (l ::l Γ) ∋ TTwk T
  lskip : Γ ∋ T → (::l Γ) ∋ TLwk T
```

We shall not repeat the cases for numbers, variables, lambda, etc as they are essentially identical to what we have seen before in Section 3.2. It remains to discuss the new expression formers, level abstraction and level application. Level abstraction Λℓ pushes a new level variable on the expression context and the type variable context. Level application •ℓ takes a level abstraction and applies it to a finite level. The resulting expression substitutes the level expression for the level variable in the body of the level abstraction.

The implication of having level abstraction and level application is that our framework has to support substitution of level expressions in level expressions and types. Level application uses $T[l']$TL for substituting level expression l' for the first level variable in T.

```
data Expr {Δ : TEnv δ} (Γ : EEnv Δ) : Type Δ l → Set where
  Λℓ_  : {T : Type (::l Δ) (Lwk l)} → Expr (::l Γ) T → Expr Γ (∀ℓ T)
  _•ℓ_ : {T : Type (::l Δ) (Lwk l)} → Expr Γ (∀ℓ T) → (l' : Lvl δ fin) → Expr Γ (T [ l' ]TL)
```

We skip the definition of variable environments ⟦_⟧Γ. They are defined by induction on expression variable contexts as a nested pair of the semantics of the respective types. The construction skips over non-value bindings tskip and lskip.

```
⟦_⟧E : {Δ : TEnv δ} {T : Type Δ l} {Γ : EEnv Δ} →
  (e : Expr Γ T) (κ : ⟦ δ ⟧δ) (η : ⟦ Δ ⟧Δ κ) (γ : ⟦ Γ ⟧Γ κ η) → ⟦ T ⟧T κ η
⟦ ‘ x     ⟧E κ η γ = lookup-γ γ x
⟦ # n     ⟧E κ η γ = n
⟦ ‘suc e ⟧E κ η γ = ℕ.suc (⟦ e ⟧E κ η γ)
⟦_⟧E {T = (_⇒_ {l₁ = l₁} {l₂ = l₂} T₁ T₂)} {Γ} (λx e) κ η γ = λ x →
  let eq₁ = ElLift≤ (⊔₁ _ _) (encode T₁ κ η) in
  let γ′ = _::γ_ {T = T₁} {Γ = Γ} (coe eq₁ x) γ in
  let eq₂ = sym (ElLift≤ (⊔₂ (⟦ l₁ ⟧L κ) _) (encode T₂ κ η)) in
  coe eq₂ (⟦ e ⟧E κ η γ′)
⟦ _·_ {l₁} {T₁ = T₁} {l₂} {T₂ = T₂} e₁ e₂ ⟧E κ η γ =
  let eq₁ = sym (ElLift≤ (⊔₁ _ (⟦ l₂ ⟧L κ)) (encode T₁ κ η)) in
  let eq₂ = ElLift≤ (⊔₂ (⟦ l₁ ⟧L κ) _) (encode T₂ κ η) in
  coe eq₂ (⟦ e₁ ⟧E κ η γ (coe eq₁ (⟦ e₂ ⟧E κ η γ)))
⟦_⟧E {Δ = Δ} {T = ∀α {l′ = l′} T} (Λ l ⇒ e) κ η γ = λ A →
  let eq = (Uⁱʳ & ext (λ j → ext (λ p → cong (λ acc → (U< {⟦ l ⟧L κ} ⦃ acc ⦄ j p)) (Acc-prop _ wf)))) in
  let η′ =  _::η_ {l = l} {Δ = Δ} {κ = κ} (coe eq A) η in
  let eq = sym (ElLift≤ (⊔₂ (⟦ ‘suc l ⟧L κ) (⟦ l′ ⟧L κ)) (encode T κ η′)) in
  coe eq (⟦ e ⟧E κ η′ γ)
⟦_⟧E {Δ = Δ} (_•_ {l′} {l = l} {T = T} e T′) κ η γ =
  let eq₁ = Uⁱʳ & ext (λ j → ext (λ p → cong (λ acc → (U< {⟦ l ⟧L κ} ⦃ acc ⦄ j p)) (Acc-prop _ wf))) in
  let eq₂ = Uⁱʳ & (ext (λ j → ext (λ p → trans (U<-compute {⟦ l ⟧L κ} {wf} {j} {p}) (sym U<-compute)))) in
  let eq₃ = cong (λ A → let η′ = _::η_ {l = l} {Δ = Δ} {κ = κ} A η in
               Elⁱʳ (Lift≤ (⊔₂ (⟦ ‘suc l ⟧L κ) _) (encode T κ η′))) (coe-coe eq₂ eq₁) in
  let eq₄ = let η′ = _::η_ {l = l} {Δ = Δ} {κ = κ} (encode T′ κ η) η in
               ElLift≤ (⊔₂ (⟦ ‘suc l ⟧L κ) _) (encode T κ η′) in
  let eq₅ = trans (trans eq₃ eq₄) ⟦[]TT⟧T in
  coe eq₅ (⟦ e ⟧E κ η γ (coe eq₂ (encode T′ κ η)))
⟦_⟧E {T = ∀ℓ {l = l} T} (Λℓ e) κ η γ = λ ℓ →
  let eq₁ = ElLift≤ (⊔₂ ω (⟦ l ⟧L κ)) (coe (cong U (⟦Lwk⟧L l κ ℓ)) (encode T (ℓ ::κ κ) η)) in
  let eq₂ = crucial l ℓ (encode T (ℓ ::κ κ) η) in
  coe (sym (trans eq₁ eq₂)) (⟦ e ⟧E (ℓ ::κ κ) η γ)
⟦ _•ℓ_ {l = l} {T = T} e l′ ⟧E κ η γ =
  let eq₁ = ElLift≤ (⊔₂ ω (⟦ l ⟧L κ)) (coe (U & ⟦Lwk⟧L l κ (⟦ l′ ⟧L′ κ)) (encode T (⟦ l′ ⟧L′ κ ::κ κ) η)) in
  let eq₂ = trans eq₁ (trans (crucial l (⟦ l′ ⟧L′ κ) (encode T (⟦ l′ ⟧L′ κ ::κ κ) η)) ⟦[]LT⟧T) in
  coe eq₂ (⟦ e ⟧E κ η γ (⟦ l′ ⟧L′ κ))
```

Fig. 1. Expression semantics

Moving to the expression semantics in Figure 1, the cases for variables and numbers are straightforward. But then things go south. The level liftings that are part of the encoding of the function type have to be reflected as transport equalities in the cases for abstraction and application. Without going into details, in each case, we have to lift the argument up to the common maximum level and then the result down to its expected level. This requirement is particularly annoying as the representations of the types are the essentially same on all levels (as evidenced by the equality ElLift≤).

Similarly, the liftings from the encoding of the universal type have to be reflected as transport equalities. For type application, we still have to deal with the same phenomenon as in Section 4, where we had to trade type substitution for type environment extension. This important detail gets lost in the noise: the "unavoidable" proof step is in eq_5 hiding in the lemma ⟦[]TT⟧T.

Finally, the cases for level abstraction and application suffer from the same problems. However, besides compensating for the level lifting, there are two unavoidable transfers involved, one is proved as the lemma crucial and the other as ⟦[]LT⟧T. crucial accounts for the fact that interpreting a weakened level expression in an extended level environment yields the same results as just interpreting the level expression in the original level environment. ⟦[]LT⟧T trades a level substitution (into a type) for a level environment extension.

5.3 Discussion

The construction of universe levels by induction-recursion is elegant and allows us to stay firmly within Set (and Set_1). However, using such an encoding for interesting examples turns out to be painful, because the encoding has to account explicitly for level subsumption aka lifting. The presence of lifting aggravates the inherent problems with transport lemmas that arise naturally in the elimination forms for universal types and level quantification.

Admittedly, the definitions in this section do not tap into the full power of the encoding. While we restrict level quantification to finite levels, the framework of Kovács [6] can easily deal with more. Due to the choice of example, we do not venture into dependent types and we do not make use of a Set-like type, all of which would be possible with the framework.

On a final note, it is possible to express the semantics of level quantification "natively" within Agda. Doing so is cumbersome and not fully general (i.e., restricted to universes up to Setω), because it requires defining a datatype that either contains a Set ℓ or a Setω. For that reason, we only defined a semantics using Kovács's construction.

6 Back to Native Semantics

One key point in the construction of an uncluttered semantics in Section 4 is the special treatment of the type Level by Agda's normalization and unification. In this section, we work in a hypothetical extension of Agda where the Level type contains limit ordinals, too. More precisely, we propose to work with ordinals less than ε_0 represented in Cantor normal form as elements of MutualOrd (cf. Section 2.2).

Even though this extension is hypothetical, we can give a taste of it by augmenting the level datatype with extra elements and laws using suitable postulates.

As our target is the same calculus as in Section 5, we can reuse the syntax along with all sorts of typing context. Of course, we redefine all environments as well as the actual semantics.

6.1 Extended Hierarchy

To construct an extended hierarchy of universes, indexed by ordinals represented by MutualOrd, we start by postulating a new (unsafe) constructor of levels, which is essentially the branching constructor of MutualOrd. It is unsafe to use this constructor directly because it does not enforce the order invariant of MutualOrd.

```
ω^_+_ : (ℓ₁ ℓ₂ : Level) → Level
```

The safe interface for constructing levels beyond ω is an injection function from MutualOrd to Level.

```
⌊_⌋ : MutualOrd → Level
⌊ 0 ⌋ = zero
⌊ ω^ l₁ + l₂ [ _ ] ⌋ = ω^ ⌊ l₁ ⌋ + ⌊ l₂ ⌋
```

Next, we postulate some equality laws for the level constructor.

```
β-suc-zero     : suc zero ≡ ω^ zero + zero -- definitional
β-suc-ω        : suc (ω^ ℓ₁ + ℓ₂) ≡ ω^ ℓ₁ + suc ℓ₂ -- definitional
distributivity : ω^ ℓ + (ℓ₁ ⊔ ℓ₂) ≡ ω^ ℓ + ℓ₁ ⊔ ω^ ℓ + ℓ₂
sub-add₁₀      : ℓ ⊔ ω^ ℓ₁ + ℓ ≡ ω^ ℓ₁ + ℓ
sub-exp₁₀      : ℓ ⊔ ω^ ℓ + ℓ₁ ≡ ω^ ℓ + ℓ₁
```

The first two laws hold definitionally for the definition of suc for MutualOrd. We proved the other three for MutualOrd. Moreover, MutualOrd with suc and ⊔ fulfills all laws for a join-semilattice with an inflationary endomorphism, which is the requirement for Agda's special treatment of Level to work.

In other words, Agda will do most of the simplification of levels with its built-in "magic", but it will need occasional help of the programmer if the above axioms are needed. To this end, we define some cast operations that manipulate the level index of the Set type.

```
cast : ∀ {ℓ₁ ℓ₂} → ℓ₁ ≡ ℓ₂ → Set ℓ₁ → Set ℓ₂
cast refl A = A

cast-intro : ∀ {ℓ₁ ℓ₂} {A : Set ℓ₁} → (eq : ℓ₁ ≡ ℓ₂) → A → cast eq A
cast-intro refl a = a

cast-elim : ∀ {ℓ₁ ℓ₂} → (eq : ℓ₁ ≡ ℓ₂) → {A : Set ℓ₁} → cast eq A → A
cast-elim refl a = a
```

Remark 2. *The cognoscenti will be reminded of the subst function from the Agda library, but this function is not applicable because it abstracts over a predicate $P : A \rightarrow$ Set ℓ, for some fixed level ℓ. In our case, a* dependent *predicate $P : (\ell :$ Level$) \rightarrow$ Set (suc ℓ) ist needed. (It is easy to extend subst accordingly.)*

6.2 Bounded Levels

Now that we squeeze a much larger fragment of the ordinals into the Level type, we need to be able to restrict level quantification to (limit) ordinals of our choosing. To this end, we define a machinery for bounding levels with an ordinal. It culminates in a record that pairs a level with the proof that it is strictly smaller than its bound.

```
record BoundedLevel (Λ : Level) : Set where
  constructor _,_
  field #_ : Level ; #<Λ : #_ < Λ
```

The crux of this seemingly obvious definition is the definition of the underlying order _ < _ on levels. We define it axiomatically, extending the standard definition of ≤ on natural numbers.[3] The order is defined as a datatype so that we can perform induction on the order. This way we avoid pattern matching on the Level datatype, which is not permitted

[3]Viz. the definition of ≤′ on natural numbers in the standard library.

```
BoundedLift : ℓ ≤ Λ → Set ℓ → Set Λ
BoundedLift (≤-id ℓ)                  A = Lift ℓ A
BoundedLift (≤-suc {ℓ₂ = ℓ₂} ℓ≤Λ)     A = Lift (suc ℓ₂) (BoundedLift ℓ≤Λ A)
BoundedLift (≤-lub ℓ₂ ℓ≤Λ)            A = Lift ℓ₂ (BoundedLift ℓ≤Λ A)
BoundedLift (≤-add {ℓ₂ = ℓ₂} ℓ₁ ℓ≤Λ) A = cast (sub-add₁₀ {ℓ = ℓ₂} {ℓ₁ = ℓ₁}) (Lift (ω^ ℓ₁ + ℓ₂) (BoundedLift ℓ≤Λ A))
BoundedLift (≤-exp {ℓ₁ = ℓ₁} ℓ₂ ℓ≤Λ) A = cast (sub-exp₁₀ {ℓ = ℓ₁} {ℓ₁ = ℓ₂}) (Lift (ω^ ℓ₁ + ℓ₂) (BoundedLift ℓ≤Λ A))
```

Fig. 2. Bounded lifting

```
⟦_⟧L′ : Lvl δ fin → ⟦ δ ⟧δ → BoundedLevel ⌊ ω ⌋        ⟦_⟧L : (l : Lvl δ any) → ⟦ δ ⟧δ → Level
⟦ ‘zero    ⟧L′ κ = zero ,                             ⟦ ‘suc l   ⟧L κ = suc (⟦ l ⟧L κ)
  ≤-exp zero 0<ω                                       ⟦ l₁ ‘⊔ l₂ ⟧L κ = ⟦ l₁ ⟧L κ ⊔ ⟦ l₂ ⟧L κ
⟦ ‘suc l   ⟧L′ κ = (suc (# ⟦ l ⟧L′ κ)) ,              ⟦ ⟨ l ⟩    ⟧L κ = # ⟦ l ⟧L′ κ
  <-suc-lim _ _ (#<Λ (⟦ l ⟧L′ κ)) (lim 0<ω)           ⟦ ‘ω       ⟧L κ = ⌊ ω ⌋
⟦ ‘ x      ⟧L′ κ = lookup-κ κ x
⟦ l₁ ‘⊔ l₂ ⟧L′ κ = # ⟦ l₁ ⟧L′ κ ⊔ # ⟦ l₂ ⟧L′ κ ,
  ≤-lublub (#<Λ (⟦ l₁ ⟧L′ κ)) (#<Λ (⟦ l₂ ⟧L′ κ))
```

Fig. 3. Semantics of level expressions

in Agda. This restriction also dictates the form of the axioms where the outcome always has the form $\ell \leq \ldots$, i.e., the left side is a variable.

```
data _≤_ : Level → Level → Set where
  ≤-id  : ∀ ℓ              → ℓ ≤ ℓ
  ≤-suc : ℓ ≤ ℓ₂           → ℓ ≤ suc ℓ₂
  ≤-lub : ∀ ℓ₁ → ℓ ≤ ℓ₂ → ℓ ≤ (ℓ₁ ⊔ ℓ₂)
  ≤-add : ∀ ℓ₁ → ℓ ≤ ℓ₂ → ℓ ≤ ω^ ℓ₁ + ℓ₂
  ≤-exp : ∀ ℓ₂ → ℓ ≤ ℓ₁ → ℓ ≤ ω^ ℓ₁ + ℓ₂

_<_ : Level → Level → Set
_<_ ℓ₁ ℓ₂ = suc ℓ₁ ≤ ℓ₂
```

```
data Lim : Level → Set where
  lim : ∀ {ℓ} → zero < ℓ → Lim (ω^ ℓ + zero)
  add : ∀ ℓ₁ → Lim ℓ₂ → Lim (ω^ ℓ₁ + ℓ₂)

≤-lublub  : ℓ₁ ≤ ℓ₃ → ℓ₂ ≤ ℓ₃ → (ℓ₁ ⊔ ℓ₂) ≤ ℓ₃
<-suc-lim : ∀ ℓ₁ ℓ₂ → ℓ₁ < ℓ₂ → Lim ℓ₂ → suc ℓ₁ < ℓ₂
```

We need two further axioms, which do not fit into the required schema for constructors of the $\_ \leq \_$ type. The axiom ≤-lublub has a maximum on the left side of the relation and <-suc-lim has a successor on the left side. The latter expresses the fact that we can always squeeze another ordinal between a limit ordinal β and some $\alpha < \beta$, namely $\alpha + 1$. To this end, the datatype Lim characterizes limit ordinals (in MutualOrd): they need to end in ω^k, for some $0 < k$.

We proved all axioms for the MutualOrd representation.

Similar to the encoding semantics in Section 5, we sometimes need to lift levels explicitly, but only in connection with level quantification. Figure 2 gives the definition, which relies on the Lift type from Agda's level infrastructure. We only show the typing part, the expression part is defined similarly. The key is that this definition is by induction on the level order $\_ \leq \_$. The definition of the order is such that the definition of lifting does *not* require pattern matching on the Level type.

```
⟦_⟧T : {Δ : TEnv δ} → (T : Type Δ l) → (κ : ⟦ δ ⟧δ) → ⟦ Δ ⟧Δ κ → Set (⟦ l ⟧L κ)
⟦ Nat       ⟧T κ η = ℕ
⟦ ‘ α       ⟧T κ η = lookup-η η α
⟦ T₁ ⇒ T₂ ⟧T κ η = ⟦ T₁ ⟧T κ η → ⟦ T₂ ⟧T κ η
⟦_⟧T {Δ = Δ} (∀α {l = l} T) κ η = ∀ A →
  let η′ = _∷η_ {l = l} {Δ = Δ} {κ = κ} A η in
  ⟦ T ⟧T κ η′
⟦_⟧T {l = l} {Δ = Δ} (∀ℓ {l = l₁} T) κ η = ∀ (ℓ : BoundedLevel ⌊ ω ⌋) →
  cast (cong (⌊ ω ⌋ ⊔_) (⟦Lwk⟧L l₁ κ ℓ)) (Lift ⌊ ω ⌋ (⟦ T ⟧T (ℓ ∷κ κ) η))
```

Fig. 4. Semantics of Types

6.3 Semantics

A level environment is still a vector of levels, but now we make it explicit that the levels are less than ω.

```
⟦_⟧δ : (δ : LEnv) → Set
⟦ [] ⟧δ = ⊤
⟦ _ ∷ δ ⟧δ = BoundedLevel ⌊ ω ⌋ × ⟦ δ ⟧δ
```

The definition of the semantics of finite level expressions (Figure 3) is slightly involved because we need to include proof that all results are bounded by ω. To this end, we need some of our axioms along with the following lemma:

```
0<ω : zero < ω^ zero + zero
0<ω = subst (suc zero ≤_) β-suc-zero (≤-id (suc zero))
```

The semantics of general level expressions is (almost) the same as before: we forget the bound when projecting from a finite level expression.

```
⟦_⟧Δ : (Δ : TEnv δ) → (κ : ⟦ δ ⟧δ) → Set (suc⨆Δ κ Δ)
⟦ []      ⟧Δ κ = ⊤
⟦ l ∷ Δ ⟧Δ κ = Set (⟦ l ⟧L κ) × ⟦ Δ ⟧Δ κ
⟦ ∷l Δ  ⟧Δ κ = ⟦ Δ ⟧Δ (drop-κ κ)
```

This definition of the type environment looks very similar to the definition from Section 4.1. The difference is that Δ now contains (finite) level expressions, which we have to interpret to get the actual element of Level to index Set. Moreover, the interpreted level is just some symbolic value that can stand for any ordinal in the range of our representation.

The semantics of types (Figure 4) has become readable, again. The cases for function types and universal types look like in Section 4.1. For level abstraction, the bound ω has to be stated and we have to lift the level of the body to (symbolic) level ω.

7 Related Work

How do other proof assistants (Coq, Lean) handle universes? Cumulativity, Impact of impredicativity

8 Conclusions

References

[1] Marc Bezem and Thierry Coquand. 2022. Loop-checking and the uniform word problem for join-semilattices with an inflationary endomorphism. *Theor. Comput. Sci.* 913 (2022), 1–7. doi:10.1016/J.TCS.2022.01.017

[2] James Chapman, Roman Kireev, Chad Nester, and Philip Wadler. 2019. System F in Agda, for Fun and Profit. In *Mathematics of Program Construction - 13th International Conference, MPC 2019 (Lecture Notes in Computer Science, Vol. 11825)*, Graham Hutton (Ed.). Springer, Porto, Portugal, 255–297. doi:10.1007/978-3-030-33636-3_10

[3] Peter Dybjer and Anton Setzer. 1999. A Finite Axiomatization of Inductive-Recursive Definitions. In *Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1581)*, Jean-Yves Girard (Ed.). Springer, 129–146. doi:10.1007/3-540-48959-2_11

[4] Fredrik Nordvall Forsberg, Chuangjie Xu, and Neil Ghani. 2020. Three equivalent ordinal notation systems in cubical Agda. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, Jasmin Blanchette and Catalin Hritcu (Eds.). ACM, 172–185. doi:10.1145/3372885.3373835

[5] Alex Hubers and J. Garrett Morris. 2023. Generic Programming with Extensible Data Types: Or, Making Ad Hoc Extensible Data Types Less Ad Hoc. *Proc. ACM Program. Lang.* 7, ICFP (2023), 356–384. doi:10.1145/3607843

[6] András Kovács. 2022. Generalized Universe Hierarchies and First-Class Universe Levels. In *30th EACSL Annual Conference on Computer Science Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany (Virtual Conference) (LIPIcs, Vol. 216)*, Florin Manea and Alex Simpson (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 28:1–28:17. doi:10.4230/LIPICS.CSL.2022.28

[7] Daniel Leivant. 1991. Finitely Stratified Polymorphism. *Inf. Comput.* 93, 1 (1991), 93–113. doi:10.1016/0890-5401(91)90053-5

[8] Conor McBride. 2015. Datatypes of Datatypes. https://www.cs.ox.ac.uk/projects/utgp/school/conor.pdf.

[9] Hannes Saffrich, Peter Thiemann, and Marius Weidner. 2024. Intrinsically Typed Syntax, a Logical Relation, and the Scourge of the Transfer Lemma. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Type-Driven Development, TyDe 2024, Milan, Italy, 6 September 2024*, Sandra Alves and Jesper Cockx (Eds.). ACM, Milan, Italy, 2–15. doi:10.1145/3678000.3678201

[10] The Agda Team. 2025. Agda Language Reference: Universe Levels. https://agda.readthedocs.io/en/stable/language/universe-levels.html.