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, **??** 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 [**?**]. It provides an abstract datatype LEVEL of universe levels along with constants ZERO, SUC, and MAX 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 : SET (SUC i) Base types inhabit the universe $Set\ zero$. Type formation handles universe levels in the same way as finitely stratified System F [**?**]. That is,

- if $A_1 : Set\ i_1$ and $A_2 : Set\ i_2$, then $A_1 \to A_2 : Set(i_1 \sqcup i_2)$;
- if $\alpha : Set\ i$ 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,

Manuscript submitted to ACM

levels ω and higher are **not** handled in a parametric way in Agda. Rather there are predefined types Set_ω, $Set_{\omega 1}$, $Set_{\omega 2}$, 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 [**?**]:

- 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 MAX: $suc(a \sqcup b)$ is the same as $suc\ a \sqcup suc\ b$.
- Neutrality of ZERO: $a \sqcup zero$ is the same as a.
- Subsumption: $a \sqcup suc\ a$ is the same as $suc\ a$.

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

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.

?] 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. **?**] constructs a very general model of cumulative universe hierarchies, which situated entirely in Set_0 and Set_1. The construction requires induction-recursion [**?**], so that the model can be constructed in Agda along with its consistency proof.

Kovács's implementation in Agda is along the lines of **?**]'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.

```
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 Uⁱʳ {i}(l : ∀ (j : Lvl) → j < i → Set) : Set        Elⁱʳ : ∀ {i l} → Uⁱʳ {i} l → Set

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

The actual universes are defined by induction on the accessibility of levels.

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

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}}}
```

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

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.

```
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.

?] 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 [**?**]. 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 **?**], who develop the syntactic metatheory of System-Fω (without stratification). It has been picked up by **?**], who give denotational and operational semantics for finitely stratified System-F and develop a logical relation for it. **?**] 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 **?**] and **?**], 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 **??**). The representation of a type variable context is thus.

```
TEnv = List Level
```

Types take a type variable context 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 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 take a variable context 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

Each denotational semantics consists of two parts: the semantics of types and the semantics of expressions.

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) $\mathsf{Set}\omega$, so that equational reasoning at the type becomes inconvenient.

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

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

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

Again, this definition lives in $\mathsf{Set}\omega$, although we might think of defining a function that computes the level of the environment from the type variable context:

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

However, Agda does not let us to define an inductive datatype where the level depends on an index type.

(3) Finally, we define it as a function 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)
```

The key to make this semantics readable lies in our choice of the Level datatype for syntactic levels. If we chose other numbers, e.g., natural numbers, for syntactic levels, the definitions for function types and universal types would become cluttered with transport lemmas (see **??**).

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 []       = zero
⌊⌋Γ (T :: Γ) = lvlOf T ⊔ ⌊⌋Γ Γ
⌊⌋Γ (ℓ ::ℓ Γ) = ⌊⌋Γ Γ
```

```
⟦_⟧Γ : (Γ : EEnv Δ) → ⟦ Δ ⟧Δ → Set (⌊⌋Γ Γ)
⟦ []     ⟧Γ η  = ⊤
⟦ T :: Γ ⟧Γ η  = ⟦ T ⟧T η × ⟦ Γ ⟧Γ η
⟦ 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 $A : \mathsf{Set}_\ell$ 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 (⟦Tid_s⟧σ _)))
                   (sym (⟦Tsub⟧T η _ T))
```

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 appears to be unavoidable. So is another equality (not shown) in the definition of $lookup-\gamma$.

The interpretation of types requires no transports because of Agda's special treatment for the normalization and unification of the Level type.

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 interpretations of level expressions, a finite one 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 of the level variables as finite levels, that is, natural numbers.

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

```
⟦_⟧L′ : Lvl δ fin → ⟦ δ ⟧δ → ℕ
⟦ ‘zero ⟧L′ κ   = ℕ.zero
⟦ ‘suc l ⟧L′ κ  = ℕ.suc (⟦ l ⟧L′ κ)
⟦ l₁ ‘⊔ l₂ ⟧L′ κ = ⟦ l₁ ⟧L′ κ ⊔ℕ ⟦ l₂ ⟧L′ κ
⟦ ‘ x ⟧L′ κ     = lookup-κ κ x

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

data TEnv : LEnv → Set where
  []   : TEnv δ
  _::_ : Lvl δ any → TEnv δ → TEnv δ
  ::l_ : TEnv δ → TEnv (tt :: δ)

data _∋_ : TEnv δ → Lvl δ any → Set where
  here  : (l :: Δ) ∋ l
  there : Δ ∋ l → (l′ :: Δ) ∋ l
  lskip : Δ ∋ l → (::l Δ) ∋ Lwk l

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)

⟦_⟧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)

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} {l′ = l′} T) κ η = Π’ (U’ (<≤-trans ℕ*ℕ.<suc (⊔₁ _ _))) λ A →
  let eq = (U^ir & ext (λ j → ext (λ p → cong (λ acc → (U< {⟦ l ⟧L κ}  acc  j p)) (Acc-prop _ wf)))) in
  let T = encode T κ (_::η_ {l = l} {Δ = Δ} {κ = κ} (coe eq A) η) in
  Lift≤ (⊔₂ _ _) T
encode (∀ℓ {l = l} T) κ η = Π’ ℕ’ λ ℓ →
  let T = coe ((cong U (⟦Lwk⟧L l κ ℓ))) (encode T (ℓ ::κ κ) η) in
  Lift≤ (⊔₂ ω _) T

⟦_⟧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)
```

```
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

data Expr {Δ : TEnv δ} (Γ : EEnv Δ) : Type Δ l → Set where
  ‘_      : Γ ∋ T → Expr Γ T
  #_      : ℕ → Expr Γ Nat
  ‘suc    : Expr Γ Nat → Expr Γ Nat
  λx_     : Expr (T :: Γ) T′ → Expr Γ (T ⇒ T′)
  Λ_⇒_    : (l : Lvl δ any) {T : Type (l :: Δ) l′} → Expr (l ::l Γ) T → Expr Γ (∀α T)
  Λℓ_     : {T : Type (::l Δ) (Lwk l)} → Expr (::l Γ) T → Expr Γ (∀ℓ T)
  _·_     : Expr Γ (T₁ ⇒ T₂) → Expr Γ T₁ → Expr Γ T₂
  _•_     : Expr Γ (∀α T) → (T′ : Type Δ l) → Expr Γ (T [ T′ ]TT)
  _•ℓ_    : {T : Type (::l Δ) (Lwk l)} → Expr Γ (∀ℓ T) → (l′ : Lvl δ fin) → Expr Γ (T [ l′ ]TL)

⟦_⟧Γ : {Δ : TEnv δ} → (Γ : EEnv Δ) → (κ : ⟦ δ ⟧δ) → ⟦ Δ ⟧Δ κ → Set
⟦ []     ⟧Γ κ η = ⊤
⟦ T :: Γ ⟧Γ κ η = ⟦ T ⟧T κ η × ⟦ Γ ⟧Γ κ η
⟦_⟧Γ {Δ = l :: Δ} (l ::l Γ) κ η = ⟦ Γ ⟧Γ κ (drop-η {l = l} {Δ = Δ} {κ = κ} η)
⟦ ::l Γ  ⟧Γ κ η = ⟦ Γ ⟧Γ (drop-κ κ) η

⟦_⟧E : {Δ : TEnv δ} {T : Type Δ l} {Γ : EEnv Δ} →
  Expr Γ T → (κ : ⟦ δ ⟧δ) (η : ⟦ Δ ⟧Δ κ) → ⟦ 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 γ′ = _::γ_ {T = T₁} {Γ = Γ} (coe (ElLift≤ (⊔₁ _ _) (encode T₁ κ η)) x) γ in
  let eq = sym (ElLift≤ (⊔₂ (⟦ l₁ ⟧L κ) (⟦ l₂ ⟧L κ)) (encode T₂ κ η)) in
  coe eq (⟦ e ⟧E κ η γ′)
⟦_⟧E {Δ = Δ} {T = ∀α {l′ = l′} T} (Λ l ⇒ e) κ η γ = λ A →
  let eq = (U^ir & 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 {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)
⟦ _·_ {l₁} {T₁ = T₁} {l₂} {T₂ = T₂} e₁ e₂ ⟧E κ η γ =
  let eq₁ = sym (ElLift≤ (⊔₁ (⟦ l₁ ⟧L κ) (⟦ l₂ ⟧L κ)) (encode T₁ κ η)) in
  let eq₂ = ElLift≤ (⊔₂ (⟦ l₁ ⟧L κ) (⟦ l₂ ⟧L κ)) (encode T₂ κ η) in
  coe eq₂ (⟦ e₁ ⟧E κ η γ (coe eq₁ (⟦ e₂ ⟧E κ η γ)))
⟦_⟧E {Δ = Δ} (_•_ {l′} {l = l} {T = T} e T′) κ η γ =
  let eq₁ = U^ir & ext (λ j → ext (λ p → cong (λ acc → (U< {⟦ l ⟧L κ} acc j p)) (Acc-prop _ wf))) in
  let eq₂ = U^ir & (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
           El^ir (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)))
⟦ _•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))
```

While the semantics of level quantification can be expressed "natively" within Agda, it is cumbersome and not fully general. For that reason, we turn to a semantics using an inductive-recursively defined universe. More precisely, we instantiate Kovács's construction suitably.

6 Related Work

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

7 Conclusions

References

[] 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

[] 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

[] 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

[] 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

[] 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

[] 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

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

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

[] 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

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

Temporary page!

LaTeX was unable to guess the total number of pages correctly. As there was some unprocessed data that should have been added to the final page this extra page has been added to receive it.

If you rerun the document (without altering it) this surplus page will go away, because LaTeX now knows how many pages to expect for this document.