Stratified Type Theory

Jonathan Chan
University of Pennsylvania, Philadelphia, USA

Stephanie Weirich
University of Pennsylvania, Philadelphia, USA

Abstract

A hierarchy of type universes is a rudimentary ingredient in the type theories of many proof assistants to prevent the logical inconsistency resulting from combining dependent functions and the type-in-type rule. In this work, we argue that a universe hierarchy is not the *only* option for a type theory with a type universe. Taking inspiration from Leivant's Stratified System F, we introduce **Stratified Type Theory** (StraTT), where rather than stratifying universes by levels, we stratify typing judgements and restrict the domain of dependent functions to strictly lower levels. Even with type-in-type, this restriction suffices to enforce consistency.

In StraTT, we consider a number of extensions beyond just stratified dependent functions. First, the subsystem subStraTT employs McBride's crude-but-effective stratification (also known as displacement) as a simple form of level polymorphism where global definitions with concrete levels can be displaced uniformly to any higher level. Second, to recover some expressivity lost due to the restriction on dependent function domains, the full StraTT includes a separate nondependent function type with a *floating* domain whose leve matches that of the overall function type. Finally, we have implemented a prototype type checker for StraTT extended with datatypes and inference for level and displacement annotations, along with a small core library.

We have proven StraTT to be type safe and subStraTT to be consistent, but consistency of the full StraTT remains an open problem, largely due to the interaction between floating functions and cumulativity of judgements. Nevertheless, we believe StraTT to be consistent, and as evidence have verified the failure of some well-known type-theoretic paradoxes using our implementation.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases type theory, dependent types, stratification

Digital Object Identifier 10.4230/LIPIcs...

Supplementary Material *Software (source code)*: `https://github.com/plclub/StraTT`
archived at `swh:1:dir:61e3b076108ebadc1a8e2fdd94cbb185f07e2483`

1 Introduction

Ever since their introduction in Martin-Löf's intuitionistic type theory (MLTT) [31], dependent type theories have included hierarchies of type universes in order to rectify the logical inconsistency of the type-in-type axiom. That is, rather than the universe $\star$ of types being its own type, these type theories have universes $\star_k$ indexed by a sequence of levels k such that the type of a universe is the universe at the next higher level.

Such a universe hierarchy is a rudimentary ingredient in many contemporary proof assistants, such as Coq [10], Agda [35], Lean [15], F* [42], and Arend [9]. For greater expressiveness, all of these also implement some sort of level polymorphism. Supporting such generality means that the proof assistant must handle level variable constraints, level expressions, or both. However, programming with and especially debugging errors involving universe levels is a common pain point among proof assistant users. So we ask: do all roads necessarily lead to level polymorphism and more generally a universe hierarchy, or are there other avenues to be taken?

© Jonathan Chan, Stephanie Weirich;
licensed under Creative Commons License CC-BY 4.0
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this work, we design **Stratified Type Theory** (StraTT) to explore one potential alternative: rather than stratifying universes into a hierarchy, we instead stratify *typing judgements* themselves by levels. This is inspired by Leivant's *Stratified System F* [26], a predicative variant of System F [19, 36]. Recall the formation rule for polymorphic type quantification in System F, given below on the left. The quantification is said to be *impredicative* because it quantifies over all types including itself, and so the type $\forall x.\, B$ itself can be substituted for x in B.

$$\frac{\Gamma, x\ \textbf{type} \vdash B\ \textbf{type}}{\Gamma \vdash \forall x.\, B\ \textbf{type}}\ \textsc{F-impredicative} \qquad \frac{\Gamma, x\ \textbf{type}\ j \vdash B\ \textbf{type}\ k \qquad j < k}{\Gamma \vdash \forall x^j.\, B\ \textbf{type}\ k}\ \textsc{F-stratified}$$

In contrast, the formation rule in Stratified System F above on the right disallows impredicativity by restricting polymorphic quantification to only types that are well formed at strictly lower stratification levels, and type well-formedness judgements are additionally indexed by a level.

To extend stratified polymorphism to dependent types, there are two ways to read this judgement form. We could interpret $\Gamma \vdash A$ **type** k as a type A living in some stratified type universe $\star_k$, which would correspond to a usual predicative type theory where $\star_j : \star_k$ when $j < k$. Alternatively, we can continue to interpret the level k as a property of the judgement and generalize it to the judgement form $\Gamma \vdash a :^k A$, where variables $x :^k A$ are also annotated with a level within the context Γ. Guided by these principles, we introduce stratified dependent function types $\Pi x :^j A.\, B$, which similarly quantify over types at the annotated level j that must be strictly lower than the overall level of the type.

To enable code reuse, in place of level polymorphism, we employ McBride's *crude-but-effective* [33]. Following Favonia, Angiuli, and Mullanix [21], we refer to this as *displacement* to prevent confusion. Given some signature Δ of global definitions, we are permitted to use any definition with all of its levels uniformly displaced upwards.

However, even in the presence of displacement, we find that stratification is sometimes *too* restrictive and can rule out terms that are otherwise typeable in an unstratified system. Therefore, StraTT includes a separate unstratified nondependent function type with a *floating* domain, so called because of its behaviour in the presence of cumulativity with respect to the levels. For a dependent function type, cumulativity can raise its overall level, but the level of the domain type remains fixed due to its level annotation. For a floating, nondependent function type whose level is raised by cumulativity, the domain type here instead floats to have the same level.

In the absence of floating nondependent functions, with only stratified dependent functions, consistency holds even with type-in-type, because the restriction on the domains of dependent functions prevents the kind of self-referential trickery that permits the usual paradoxes. However, we haven't yet proven consistency with the inclusion of floating nondependent functions; the primary barrier is the covariant behaviour of the floating domain with respect to levels, which is unusual for function types. Even so, we have found it impossible to encode some well-known type-theoretic paradoxes, leading us to believe that consistency *does* hold, which would make the system suitable as a foundation for theorem proving.

These features form the basis of StraTT, and our contributions are as follows:

- A subsystem subStraTT, featuring only stratified dependent functions and displacement, which is then extended to the full StraTT with floating nondependent functions. $\hookrightarrow$ Section 2

- A number of examples to demonstrate the expressivity of StraTT and especially to motivate floating functions. ↪ Section 3
- Two major metatheorems: logical consistency for subStraTT, which is mechanized in Agda, and type safety for StraTT, which is mechanized in Coq. Consistency for the full StraTT remains an open problem. ↪ Section 4
- A prototype implementation of a type checker, which extends StraTT to include datatypes to demonstrate the effectiveness of stratification and displacement in practical dependently-typed programming. ↪ Section 5

We discuss potential avenues for proving consistency of the full StraTT and compare the useability of its design to existing proof assistants in terms of working with universe levels in Section 6, and conclude in Section 7. Our Agda and Coq mechanizations along with the prototype implementation are available in the supplementary material. Where lemmas and theorems are first introduced, we include a footnote indicating the corresponding source file and lemma name in the development.

2 Stratified Type Theory

In this section, we present Stratified Type Theory in two parts. First is the subsystem subStraTT, which contains the two core features of stratified dependent function types and global definitions with level displacement. We then extend it to the full StraTT by adding floating nondependent function types. As the system is fairly small with few parts, we delay illustrative examples to Section 3, and begin with the formal description.

2.1 The subsystem subStraTT

The subsystem subStraTT is a cumulative, extrinsic type theory with types à la Russell, a single type universe, dependent functions, an empty type, and global definitions. The most significant difference between subStraTT and other type theories with these features is the annotation of the typing judgement with a level in place of universes in a hierarchy. We use the naturals and their usual strict order and addition operation for our levels, but they should be generalizable to any displacement algebra [21]. The syntax is given below, with x, y, z for variable and constant names and i, j, k for levels.

$$a, b, c, A, B, C ::= \star \mid x \mid x^i \mid \Pi x :^j A.\, B \mid \lambda x.\, b \mid b\ a \mid \bot \mid \mathsf{absurd}(b)$$

The typing judgement has the form $\boxed{\Delta; \Gamma \vdash a :^k A}$; its typing rules are given in Figure 1. The judgement states that term a is well typed at level k with type A under the context Γ and signature Δ. A context consists of declarations $x :^k A$ of variables x of type A at level k; variables represent locations where an entire typing derivation may be substituted into the term, so they also need level annotations. A signature consists of global definitions $x :^k A \coloneqq a$ of constants x of type A definitionally equal to a at level k; they represent complete typing derivations that will eventually be substituted into the term.

Because stratified judgements replace stratified universes, the type of the type universe $\star$ is itself at any level in rule DT-Type. Stratification is enforced in dependent function types in rule DT-Pi: the domain type must be well typed at a strictly smaller level relative to the codomain type and the overall function type. Similarly, in rule DT-AbsTy, the body of a dependent function is well typed when its argument and its type are well typed at a strictly smaller level, and by rule DT-AppTy, a dependent function can only be applied to an argument at the strictly smaller domain level.

$\boxed{\Delta; \Gamma \vdash a :^k A}$ (*Typing*)

$$\text{DT-Type}\ \frac{\Delta \vdash \Gamma}{\Delta; \Gamma \vdash \star :^k \star} \qquad \text{DT-Pi}\ \frac{\begin{array}{c}\Delta; \Gamma \vdash A :^j \star \\ \Delta; \Gamma, x :^j A \vdash B :^k \star \\ j < k\end{array}}{\Delta; \Gamma \vdash \Pi x :^j A.\, B :^k \star} \qquad \text{DT-AbsTy}\ \frac{\begin{array}{c}\Delta; \Gamma \vdash A :^j \star \\ \Delta; \Gamma, x :^j A \vdash b :^k B \\ j < k\end{array}}{\Delta; \Gamma \vdash \lambda x.\, b :^k \Pi x :^j A.\, B}$$

$$\text{DT-AppTy}\ \frac{\begin{array}{c}\Delta; \Gamma \vdash b :^k \Pi x :^j A.\, B \\ \Delta; \Gamma \vdash a :^j A \qquad j < k\end{array}}{\Delta; \Gamma \vdash b\ a :^k B\{a/x\}} \qquad \text{DT-Var}\ \frac{\begin{array}{c}x :^j A \in \Gamma \\ \Delta \vdash \Gamma \qquad j \leq k\end{array}}{\Delta; \Gamma \vdash x :^k A} \qquad \text{DT-Const}\ \frac{\begin{array}{c}x :^j A \coloneqq a \in \Delta \qquad \Delta \vdash \Gamma \\ \vdash \Delta \qquad i + j \leq k\end{array}}{\Delta; \Gamma \vdash x^i :^k A^{+i}}$$

$$\text{DT-Bottom}\ \frac{\Delta \vdash \Gamma}{\Delta; \Gamma \vdash \bot :^k \star} \qquad \text{DT-Absurd}\ \frac{\begin{array}{c}\Delta; \Gamma \vdash A :^k \star \\ \Delta; \Gamma \vdash b :^k \bot\end{array}}{\Delta; \Gamma \vdash \mathsf{absurd}(b) :^k A} \qquad \text{DT-Conv}\ \frac{\begin{array}{c}\Delta; \Gamma \vdash a :^k A \\ \Delta; \Gamma \vdash B :^k \star \\ \Delta \vdash A \equiv B\end{array}}{\Delta; \Gamma \vdash a :^k B}$$

Figure 1 Typing rules (subStraTT)

Note that the level annotation on dependent function types is necessary for consistency. Informally, suppose we have some unannotated type $\Pi X : \star.\, B$ and a function of this type, both at level 1. By cumulativity, we can raise the level of the function to 2, then apply it to its own type $\Pi X : \star.\, B$. In short, impredicativity is reintroduced, and stratification defeated.

Rules DT-Bottom and DT-Absurd are the uninhabited type and its eliminator, respectively. It should be consistent to eliminate a falsehood into any level, including lower levels, but when viewed bottom-up, the level of the conclusion represents the level of the entire derivation tree, or the level of all the pieces used to construct the tree, so it wouldn't make sense to allow premises at higher levels.

In rules DT-Var and DT-Const, variables and constants at level j can be used at any larger level k, which we refer to as subsumption. This permits the following admissible cumulativity lemma, allowing entire derivations to be used at larger levels.

▶ **Lemma 1** (Cumulativity).[1] *If $\Delta; \Gamma \vdash a :^j A$ and $j \leq k$ then $\Delta; \Gamma \vdash a :^k A$.*

Constants are also annotated with a superscript indicating how much they're displaced by. If a constant x is defined with a type A, we're permitted to use x^i as an element of type A but with all of its levels incremented by i. The metafunction a^{+i} performs this increment in the term a, defined recursively with $(\Pi x :^j A.\, B)^{+i} = \Pi x :^{i+j} A^{+i}.\, B^{+i}$ and $(x^j)^{+i} = x^{i+j}$. Constants come from signatures and variables from contexts, whose key formation rules for the judgements $\boxed{\vdash \Delta}$ and $\boxed{\Delta \vdash \Gamma}$ respectively are given below.

$$\text{D-Cons}\ \frac{\begin{array}{c}\vdash \Delta \qquad \Delta; \varnothing \vdash A :^k \star \\ \Delta; \varnothing \vdash a :^k A \qquad x \notin \mathsf{dom}\,\Delta\end{array}}{\vdash \Delta, x :^k A \coloneqq a} \qquad \text{DG-Cons}\ \frac{\begin{array}{c}\Delta \vdash \Gamma \\ \Delta; \Gamma \vdash A :^k \star \qquad x \notin \mathsf{dom}\,\Gamma \qquad x \notin \mathsf{dom}\,\Delta\end{array}}{\Delta \vdash \Gamma, x :^k A}$$

[1] `coq/restrict.v:DTyping_cumul`

In rule DT-Conv, we use an untyped definitional equality $\boxed{\Delta \vdash a \equiv b}$ that is reflexive, symmetric, transitive, and congruent, and includes β-equivalence for functions and δ-equivalence of constants x with their definitions. When a constant is displaced as x^i, we must also increment the level annotations in their definitions by i. Below are the rules for β- and δ-equivalence; the remaining rules can be found in Appendix A.

$$\text{DE-Beta}\quad \frac{}{\Delta \vdash (\lambda x.\, b)\ a \equiv b\{a/x\}} \qquad \text{DE-Delta}\quad \frac{x :^k A \coloneqq a \in \Delta}{\Delta \vdash x^i \equiv a^{+i}}$$

Given a well-typed, locally-closed term $\Delta; \varnothing \vdash a :^k A$, the entire derivation itself can be displaced upwards by some increment i. This lemma differs from cumulativity, since the level annotations in the term and its type are displaced as well, not just that of the judgement.

▶ **Lemma 2** (Displaceability (empty context)).[2] *If* $\Delta; \varnothing \vdash a :^k A$ *then* $\Delta; \varnothing \vdash a^{+i} :^{i+k} A^{+i}$.

With $x :^k A \coloneqq a$ in the signature, x^i is definitionally equal to a^{+i}, so this lemma justifies rule DT-Const, which would give this displaced constant the type A^{+i}.

2.2 Floating functions

As we'll see in the next section, subStraTT alone is insufficiently expressive, with some examples being unexpectedly untypeable and others being simply clunky to work with as a result of the strict restriction on function domains. The full StraTT system therefore extends the subsystem with a separate nondependent function type, written $A \to B$, whose domain doesn't have the same restriction.

$$\text{DT-Arrow}\quad \frac{\Delta; \Gamma \vdash A :^k \star \qquad \Delta; \Gamma \vdash B :^k \star}{\Delta; \Gamma \vdash A \to B :^k \star} \qquad \text{DT-AbsTm}\quad \frac{\Delta; \Gamma \vdash A :^k \star \qquad \Delta; \Gamma \vdash B :^k \star \qquad \Delta; \Gamma, x :^k A \vdash b :^k B}{\Delta; \Gamma \vdash \lambda x.\, b :^k A \to B} \qquad \text{DT-AppTm}\quad \frac{\Delta; \Gamma \vdash b :^k A \to B \qquad \Delta; \Gamma \vdash a :^k A}{\Delta; \Gamma \vdash b\ a :^k B}$$

Figure 2 Typing rules (floating functions)

The typing rules for nondependent function types, functions, and application are given in Figure 2. The domain, codomain, and entire nondependent function type are all typed at the same level. Functions take arguments of the same level as their bodies, and are thus applied to arguments of the same level.

This distinction between stratified dependent and unstratified nondependent functions corresponds closely to Stratified System F: type polymorphism is syntactically distinct from ordinary function types, and the former forces the codomain to be a higher level while the latter doesn't. From the perspective of Stratified System F, the dependent types of StraTT generalize stratified type polymorphism over types to include term polymorphism.

We say that the domain of these nondependent function types *floats* because unlike the stratified dependent function types, it isn't fixed to some particular level. The interaction between floating functions and cumulativity is where this becomes interesting. Given a function f of type $A \to B$ at level j, by cumulativity, it remains well typed with the same type at any level $k \geq j$. The level of the domain floats up from j to match the function at k,

[2] `coq/incr.v:DTyping_incr`

in the sense that f can be applied to an argument of type A at any greater level k. This is unusual because the domain isn't contravariant with respect to the ordering on the levels as we might expect, and is why, as we'll see shortly, the proof of consistency in Section 4.1 can't be straightforwardly extended to accommodate floating function types.

3 Examples

3.1 The identity function

In the following examples, we demonstrate why floating functions are essential. Below on the left is one way we could assign a type to the type-polymorphic identity function. For concision, we use a pattern syntax when defining global functions and place function arguments to the left of the definition. (The subscript is part of the constant name.)

$$\mathsf{id}_0 :^1 \Pi X :^0 \star. \Pi x :^0 X. X \qquad \mathsf{id} :^1 \Pi X :^0 \star. X \to X$$
$$\mathsf{id}_0\ X\ x \coloneqq x \qquad \mathsf{id}\ X\ x \coloneqq x$$

Stratification enforces that the codomain of the function type and the function body have a higher level than that of the domain and the argument, so the overall identity function is well typed at level 1. While x and X have level 0 in the context of the body, by subsumption, we can use x at level 1 in the body as required.

Alternatively, since the return type doesn't depend on the second argument, we can use a floating function type instead, given above on the right. Since we still have a dependent type quantification, the function $X \to X$ is still typed at level 1. This means that x now has level 1 directly rather than through subsumption.

So far, there's no reason to pick one over the other, so let's look at a more involved example: applying an identity function to itself. This is possible due to cumulativity, and we'll follow the corresponding Coq example below.

```
Universes u0 u1.
Constraint u0 < u1.
Definition idid1 (id : forall (X : Type@{u1}), X -> X) :
  forall (X : Type@{u0}), X -> X :=
  id (forall (X : Type@{u0}), X -> X) (fun X => id X).
```

Here, since `forall (X : Type@{u0}), X -> X` can be assigned type `Type@{u1}`, it can be applied as the first argument to `id`. While `id` itself doesn't have this type, we can η-expand it to a function that does, since `Type@{u0}` is a subtype of `Type@{u1}`, so `X` can be passed to `id`.

If we try to write the analogous definition in subStraTT without using floating functions, we find that it doesn't type check! The problematic subterm is underlined in red below.

$$\mathsf{idid}_1 :^3 \Pi id :^2 (\Pi X :^1 \star. \Pi x :^1 X. X). \Pi X :^0 \star. \Pi x :^0 X. X$$
$$\mathsf{idid}_1\ id \coloneqq id\ (\Pi X :^0 \star. \Pi x :^0 X. X)\ \underline{(\lambda X. \lambda x. id\ X\ x)}$$

After η-expansion, $\lambda X. \lambda x. id\ X\ x$ has the correct type $\Pi X :^0 \star. \Pi x :^0 X. X$, but only at level 2, since that's the level of id itself. Meanwhile, the second argument of id expects an argument of that type but *at level 1*. We can't just raise the level annotation for that argument to 2, either, since that would raise the level of id to 3.

If we instead use floating functions for the nondependent argument, the analogous definition then *does* type check, since the second argument of type X can now be at level 2.

$$\mathsf{idid}_1 :^2 (\Pi X :^1 \star. X \to X) \to \Pi X :^0 \star. X \to X$$

$\mathsf{idid}_1\ id \coloneqq id\ (\Pi X :^0 \star.\ X \to X)\ (\lambda X.\ id\ X)$

This definition of idid1 is now pretty much shaped the same as the Coq version, only with level annotations on domains where Coq has the corresponding level annotations on `Type`. If we were to turn on universe polymorphism in Coq, it would achieve the same kind of expressivity of being able to displace idid2 in StraTT. The only difference is that while Coq merely enforces a strict inequality constraint between the levels, in StraTT the levels annotations are concrete, so even with displacement, the distance between the two levels in the type is always 1.

As an additional remark, even with floating functions, repeatedly nesting identity function self-applications is one way to non-trivially force the level to increase. The following definitions continue the pattern from $idid_1$, which in the untyped setting would correspond to $\lambda id.\ id\ id$, $\lambda id.\ id\ (\lambda id.\ id\ id)\ id$, $\lambda id.\ id\ (\lambda id.\ id\ (\lambda id.\ id\ id)\ id)\ id$, and so on.

$$\mathsf{idid}_2 :^3 (\Pi X :^2 \star.\ X \to X) \to \Pi X :^0 \star.\ X \to X$$
$$\mathsf{idid}_2\ id \coloneqq id\ ((\Pi X :^1 \star.\ X \to X) \to \Pi X :^0 \star.\ X \to X)\ \mathsf{idid}_1\ (\lambda X.\ \lambda x.\ id\ X\ x)$$
$$\mathsf{idid}_3 :^4 (\Pi X :^3 \star.\ X \to X) \to \Pi X :^0 \star.\ X \to X$$
$$\mathsf{idid}_3\ id \coloneqq id\ ((\Pi X :^2 \star.\ X \to X) \to \Pi X :^0 \star.\ X \to X)\ \mathsf{idid}_2\ (\lambda X.\ \lambda x.\ id\ X\ x)$$

All of $idid_1\ (\lambda X.\ \lambda x.\ x)$, $idid_2\ (\lambda X.\ \lambda x.\ x)$, and $idid_3\ (\lambda X.\ \lambda x.\ x)$ reduce to $\lambda X.\ \lambda x.\ x$.

3.2 Decidable types

The following example demonstrates a more substantial use of StraTT in the form of type constructors as floating functions and how they interact with cumulativity. Later in Section 5 we'll consider datatypes with parameters, but for now, consider the following Church encoding [7] of decidable types, which additionally uses negation defined as implication into the empty type.

$$\mathsf{neg} :^0 \star \to \star \qquad \mathsf{yes} :^1 \Pi X :^0 \star.\ X \to \mathsf{Dec}\ X$$
$$\mathsf{neg}\ X \coloneqq X \to \bot \qquad \mathsf{yes}\ X\ x \coloneqq \lambda Z.\ \lambda f.\ \lambda g.\ f\ x$$
$$\mathsf{Dec} :^1 \star \to \star \qquad \mathsf{no} :^1 \Pi X :^0 \star.\ \mathsf{neg}\ X \to \mathsf{Dec}\ X$$
$$\mathsf{Dec}\ X \coloneqq \Pi Z :^0 \star.\ (X \to Z) \to (\mathsf{neg}\ X \to Z) \to Z \qquad \mathsf{no}\ X\ nx \coloneqq \lambda Z.\ \lambda f.\ \lambda g.\ g\ nx$$

The yes X constructor decides X by a witness, while the no X constructor decides X by its refutation. We're able to show that deciding a given type is irrefutable[3]

$$\mathsf{irrDec} : \Pi X :^0 \star.\ \mathsf{neg}\ (\mathsf{neg}\ (\mathsf{Dec}\ X))$$
$$\mathsf{irrDec}\ X\ ndec \coloneqq ndec\ (\mathsf{no}\ X\ (\lambda x.\ ndec\ (\mathsf{yes}\ X\ x)))$$

The same exercise of trying to define neg and Dec using only dependent functions and not floating functions to the same effect of no longer being able to type check irrDec, even if we allow ourselves to use displacement. More interestingly, let's now compare these definitions to the corresponding ones in Agda.

```
{-# OPTIONS --cumulativity #-}
open import Agda.Primitive using (lzero ; lsuc)
```

[3] Note this differs from irrefutability of the law of excluded middle, $\mathsf{neg}\ (\mathsf{neg}\ (\Pi X :^0 \star.\ \mathsf{Dec}\ X))$, which cannot be proven constructively.

```
open import Data.Empty using (⊥)
neg : ∀ ℓ → Set ℓ → Set ℓ
neg ℓ X = X → ⊥
Dec : ∀ ℓ → Set (lsuc ℓ) → Set (lsuc ℓ)
Dec ℓ X = (Z : Set ℓ) → (X → Z) → (neg (lsuc ℓ) X → Z) → Z
yes : ∀ ℓ (X : Set ℓ) → X → Dec ℓ X
yes ℓ X x = λ Z f g → f x
no : ∀ ℓ (X : Set ℓ) → neg ℓ X → Dec ℓ X
no ℓ X nx = λ Z f g → g nx
```

They must all be universe polymorphic to capture the expressivity of floating functions. For instance, to talk about the negation of a type at level 1, by cumulativity it suffices to use neg (without displacement!) in StraTT, but we must use `neg (lsuc lzero)` in Agda. Effectively, the StraTT type $\star \to \star$ represents not merely `Set → Set` but, by cumulativity, all types `Set ℓ → Set ℓ` for every `ℓ`.

3.3 Leibniz equality

Although nondependent functions can often benefit from a floating domain, sometimes we don't want the domain to float. Here, we turn to a simple application of dependent types with Leibniz equality [25, 30] to demonstrate a situation where the level of the domain needs to be fixed to something strictly smaller than that of the codomain even when the codomain doesn't depend on the function argument.

$$\mathsf{eq} :^1 \Pi X :^0 \star .\, X \to X \to \star \qquad \mathsf{refl} :^1 \Pi X :^0 \star .\, \Pi x :^0 X .\, \mathsf{eq}\ X\ x\ x$$
$$\mathsf{eq}\ X\ x\ y \coloneqq \Pi P :^0 X \to \star .\, P\ x \to P\ y \qquad \mathsf{refl}\ X\ x\ P\ px \coloneqq px$$

An equality $\mathsf{eq}\ A\ a\ b$ states that two terms are equal if given any predicate P, a proof of $P\ a$ yields a proof of $P\ b$; in other words, a and b are indiscernible. The proof of reflexivity of Leibniz equality should be unsurprising.

We might try to define a predicate stating that a given type X is a mere proposition, *i.e.* that all of its inhabitants are equal, and give it a nondependent function type.

$$\mathsf{isProp} :^0 \star \to \star$$
$$\mathsf{isProp}\ X \coloneqq \underline{\Pi x :^0 X .\, \Pi y :^0 X .\, \mathsf{eq}\ X\ x\ y}$$

But this doesn't type check, since the body contains an equality over elements of X, which necessarily has level 1 rather than the expected level 0. We must assign isProp a stratified function type, given below on the left; informally, stratification propagates dependency information not only from the codomain, but also from the function body.

$$\mathsf{isProp} :^1 \Pi X :^0 \star .\, \star \qquad \mathsf{isSet} :^2 \Pi X :^0 \star .\, \star$$
$$\mathsf{isProp}\ X \coloneqq \Pi x :^0 X .\, \Pi y :^0 X .\, \mathsf{eq}\ X\ x\ y \qquad \mathsf{isSet}\ X \coloneqq \Pi x :^0 X .\, \Pi y :^0 X .\, \mathsf{isProp}^1\ (\mathsf{eq}\ X\ x\ y)$$

Going one further, we define above on the right a predicate isSet stating that X is an h-set [44], or that its equalities are mere propositions, by using a displaced isProp so that we can reuse the definition at a higher level; here, isProp^1 now has type $\Pi X :^1 \star .\, \star$ at level 2. Once again, despite the type of isSet not being an actual dependent function type, here we need to fix the level of the domain.

4 Metatheory

4.1 Consistency of subStraTT

We use Agda to mechanize a proof of logical consistency — that no closed inhabitant of the empty type exists — for subStraTT, which excludes floating nondependent functions. For simplicity, the mechanization also excludes global definitions and displaced constants, which shouldn't affect consistency: if there is a closed inhabitant of the empty type that uses global definitions, then there is a closed inhabitant of the empty type under the empty signature by inlining all global definitions. The proof files are available at `https://github.com/plclub/StraTT` under the `agda/` directory. The only axiom we use is function extensionality.[4]

The core construction of the consistency proof is a three-place logical relation $\boxed{a \in [\![A]\!]_k}$ among a term, its type, and its level, which we would aspirationally like to define as follows, using $\mathbf{0}$ for falsehood, $\mathbf{1}$ for truthhood, $\wedge$ for conjunction, $\longrightarrow$ for implication, and $\forall$ and $\exists$ for universal and existential quantification in our working metatheory.

$$\begin{array}{ll}
\star \in [\![\star]\!]_k \triangleq \mathbf{1} & \Pi x\!:^j A.\, B \in [\![\star]\!]_k \triangleq j < k \wedge A \in [\![\star]\!]_j \wedge (\forall y.\, y \in [\![A]\!]_j \longrightarrow B\{y/x\} \in [\![\star]\!]_k) \\
\bot \in [\![\star]\!]_k \triangleq \mathbf{1} & f \in [\![\Pi x\!:^j A.\, B]\!]_k \triangleq \forall y.\, y \in [\![A]\!]_j \longrightarrow f\ y \in [\![B\{y/x\}]\!]_k \\
a \in [\![\bot]\!]_k \triangleq \mathbf{0} & a \in [\![A]\!]_k \triangleq \exists B.\, A \equiv B \wedge a \in [\![B]\!]_k
\end{array}$$

However, this definition isn't necessarily well formed. It isn't defined recursively on the structure of the terms or the types, because in the cases involving dependent functions, we need to talk about the substitution $B\{y/x\}$. It isn't defined inductively, either, because again in the dependent function case, the inductive itself appears to the left of an implication as $y \in [\![A]\!]_j$, making the inductive definition non-strictly-positive.

The solution is to define the logical relation as an inductive–recursive definition [17]. This design is adapted from a concise proof of consistency for MLTT in Coq by Liu [28], which uses an impredicative encoding in place of induction–recursion. This is a simplified and pared down adaptation of a proof of decidability of conversion for MLTT in Coq by Adjedj, Lennon-Bertrand, Maillard, Pédrot, and Pujet [2], which in turn uses a predicative encoding to adapt a proof of decidability of conversion for MLTT in Agda by Abel, Öhman, and Vezzosi [1] that uses induction–recursion.

Below is a sketch of the inductive–recursive definition, which splits the logical relation into two parts: an inductive predicate on types and their levels $\boxed{[\![A]\!]_k}$, and relation between types and terms defined recursively on the predicate on the type, which we continue to write as $\boxed{a \in [\![A]\!]_k}$.

$$\frac{}{[\![\star]\!]_k} \qquad \frac{}{[\![\bot]\!]_k} \qquad \frac{j < k \qquad [\![A]\!]_j \qquad \forall y.\, y \in [\![A]\!]_j \longrightarrow [\![B\{y/x\}]\!]_k}{[\![\Pi x\!:^j A.\, B]\!]_k} \qquad \frac{A \Rightarrow B \qquad [\![B]\!]_k}{[\![A]\!]_k}$$

$$\begin{array}{ll}
A \in [\![\star]\!]_k \triangleq [\![A]\!]_k & f \in [\![\Pi x\!:^j A.\, B]\!]_k \triangleq \forall y.\, y \in [\![A]\!]_j \longrightarrow f\ y \in [\![B\{y/x\}]\!]_k \\
a \in [\![\bot]\!]_k \triangleq \mathbf{0} & a \in [\![A]\!]_k \triangleq a \in [\![B]\!]_k \quad (\textit{where } A \Rightarrow B)
\end{array}$$

In the last inductive rule, in place of $A \equiv B$, we instead use parallel reduction $\boxed{A \Rightarrow B}$, which is a reduction relation describing all visible reductions being performed in parallel from the inside out. This is justified by the following lemma, where $\boxed{A \Rightarrow^* B}$ is the reflexive, transitive closure of $A \Rightarrow B$.

[4] `agda/accessibility.agda:funext,funext'`

▶ **Lemma 3** (Implementation of definitional equality).[5] *$A \equiv B$ iff there exists some C such that $A \Rightarrow^* C \mathrel{{}^*{\Leftarrow}} B$, which we write as* $\boxed{A \Leftrightarrow B}$.

Even now, this inductive–recursive definition is *still* not well formed. In particular, in the inductive rule for dependent functions, if A is $\star$, then by the recursive case for the universe, $[\![y]\!]_j$ could again appear to the left of an implication. However, we know that $j < k$, which we can exploit to stratify the logical relation just as we stratify typing judgements. We do so by parametrizing each logical relation at level k by an abstract logical relation defined at all strictly lower levels $j < k$, then at the end tying the knot by instantiating them via well-founded induction on levels. This technique is adapted from an Agda model of a universe hierarchy by Kovács [24], which originates from McBride's redundancy-free construction of a universe hierarchy [34, Section 6.3.1]. As the constructions are now fairly involved, we defer to the proof file[6] for the full definitions, in particular `U` for the inductive predicate and `el` for the recursive relation. For the purposes of exposition, we continue to use the old notation.

Because the logical relation only handles closed terms, we deal with contexts and simultaneous substitutions σ separately by relating the two via yet another inductive–recursive definition, with a predicate on contexts $\boxed{[\![\Gamma]\!]}$ and a relation between substitutions and contexts $\boxed{\sigma \in [\![\Gamma]\!]}$. Here, $A\{\sigma\}$ denotes applying the substitution σ to the term A, and $\sigma[x]$ denotes the term which σ substitutes for x.[7]

$$\frac{}{[\![\varnothing]\!]} \qquad \frac{[\![\Gamma]\!] \qquad \forall \sigma.\, \sigma \in [\![\Gamma]\!] \longrightarrow [\![A\{\sigma\}]\!]_k}{[\![\Gamma, x :^k A]\!]}$$

$$\sigma \in [\![\varnothing]\!] \triangleq \mathbf{1}$$
$$\sigma \in [\![\Gamma, x :^k A]\!] \triangleq \sigma \in [\![\Gamma]\!] \wedge \sigma[x] \in [\![A\{\sigma\}]\!]_k$$

The most important lemmas that are needed are semantic cumulativity, semantic conversion, and backward preservation.

▶ **Lemma 4** (Cumulativity).[8] *If $j < k$ and $[\![A]\!]_j$ then $[\![A]\!]_k$, and if $a \in [\![A]\!]_j$ then $a \in [\![A]\!]_k$.*

▶ **Lemma 5** (Conversion).[9] *If $A \Leftrightarrow B$ and $[\![A]\!]_k$ then $[\![B]\!]_k$, and if $a \in [\![A]\!]_k$ then $a \in [\![B]\!]_k$.*

▶ **Lemma 6** (Backward preservation).[10] *If $a \Rightarrow^* b$ and $b \in [\![A]\!]_k$ then $a \in [\![A]\!]_k$.*

We can now prove the fundamental theorem of soundness of typing judgements with respect to the logical relation by induction on typing derivations, and consistency follows as a corollary.

▶ **Theorem 7** (Soundness).[11] *Suppose $[\![\Gamma]\!]$ and $\sigma \in [\![\Gamma]\!]$. If $\Gamma \vdash a :^k A$, then $[\![A\{\sigma\}]\!]_k$ and $a\{\sigma\} \in [\![A\{\sigma\}]\!]_k$.*

▶ **Corollary 8** (Consistency).[12] *There are no b, k such that $\varnothing \vdash b :^k \bot$.*

4.1.1 The problem with floating functions

This proof can't be extended to the full StraTT. While floating nondependent function types can be added to the logical relation directly as below, cumulativity will no longer hold.

$$\frac{[\![A]\!]_k \qquad [\![B]\!]_k}{[\![A \to B]\!]_k} \qquad\qquad f \in [\![A \to B]\!]_k \triangleq \forall x.\, x \in [\![A]\!]_k \longrightarrow f\ x \in [\![B]\!]_k$$

---

[5] `agda/typing.agda:≈-⇔` [6] `agda/semantics.agda` [7] The mechanization uses de Bruijn indexing; various index-shifting operations on substitutions are omitted for concision. [8] `agda/semantics.agda:cumU,cumEl` [9] `agda/semantics.agda:⇔-U,⇔-el` [10] `agda/semantics.agda:⇒⋆-el` [11] `agda/soundness.agda:soundness` [12] `agda/consistency.agda:consistency`

In particular, given $f \in [\![A \to B]\!]_j$, when trying to show $f \in [\![A \to B]\!]_k$, we have by definition $\forall x.\, x \in [\![A]\!]_j \longrightarrow f\ x \in [\![B]\!]_j$, a term x, and $x \in [\![A]\!]_k$, but no way to cast the latter into $x \in [\![A]\!]_j$ to obtain $f\ x \in [\![B]\!]_k$ as desired via the induction hypothesis, because such a cast would go *downwards* from a higher level k to a lower level j, rather than the other way around as provided by the induction hypothesis. Trying to incorporate the desired property into the relation, perhaps by defining it as $\forall \ell \geq k.\, \forall x.\, x \in [\![A]\!]_\ell \longrightarrow f\ x \in [\![B]\!]_k$, would break the careful stratification of the logical relation that we've set up.

4.2 Type safety of StraTT

While we haven't yet proven its consistency, we have proven type safety of the full StraTT. We use Coq to mechanize the syntactic metatheory of the typing, context formation, and signature formation judgements of StraTT, recalling that this covers all of stratified dependent functions, floating nondependent functions, and displaced constants. We also use Ott [39] along with the Coq tools LNgen [3] and Metalib [4] to represent syntax and judgements and to handle their locally-nameless representation in Coq. The proof scripts are available at `https://github.com/plclub/StraTT` under the `coq/` directory.

We begin with some basic common properties of type systems, namely weakening, substitution, and regularity lemmas, as well as a generalized displaceability lemma that's simple to show. Next, we introduce a notion of *restriction*, which formalizes the idea that lower judgements can't depend on higher ones, along with a notion of *restricted floating*, which is crucial for proving that floating function types are *syntactically* cumulative. Only then are we able to prove type safety.

As we haven't mechanized the syntactic metatheory of definitional equality $\Delta \vdash A \equiv B$, we state as axioms some standard, provable properties [5], which are orthogonal to stratification and only used in the final proof of type safety. The equivalent lemmas for subStraTT, however, have been mechanized in Agda[13] as part of the consistency proof.

▶ **Axiom 9** (Function type injectivity).[14] *If $\Delta \vdash A_1 \to B_1 \equiv A_2 \to B_2$ then $\Delta \vdash A_1 \equiv A_2$ and $\Delta \vdash B_1 \equiv B_2$; if $\Pi x {:}^{j_1} A_1.\, B_1 \equiv \Pi x {:}^{j_2} A_2.\, B_2$ then $\Delta \vdash A_1 \equiv A_2$, $j_1 = j_2$, and $\Delta \vdash B_1 \equiv B_2$.*

▶ **Axiom 10** (Consistency of definitional equality).[15] *If $\Delta \vdash A \equiv B$ then A and B do not have different head forms.*

4.2.1 Basic properties

We can extend the ordering between levels $j \leq k$ to an ordering between contexts $\boxed{\Gamma_1 \leq \Gamma_2}$; that is, if $j \leq k$, then $\Gamma, x :^j A \leq \Gamma, x :^k A$. At the same time, we also incorporate the idea of weakening into this relation, so $\Gamma, x :^k A \leq \Gamma$. Stronger contexts have higher levels and fewer assumptions. This ordering is contravariant in the typing judgement: we can lower the context without destroying typeability. This result subsumes a standard weakening lemma.

▶ **Lemma 11** (Weakening).[16] *If $\Delta; \Gamma \vdash a :^k A$ and $\Delta \vdash \Gamma'$ and $\Gamma' \leq \Gamma$ then $\Delta; \Gamma' \vdash a :^k A$.*

The substitution lemma reflects the idea that an assumption $x :^k B$ is a hypothetical judgement. The variable x stands for any typing derivation of the appropriate type and level.

▶ **Lemma 12** (Substitution).[17] *If $\Delta; \Gamma_1, x :^j B, \Gamma_2 \vdash a :^k A$ and $\Delta; \Gamma_1 \vdash b :^j B$ then $\Delta; \Gamma_1, \Gamma_2\{b/x\} \vdash a\{b/x\} :^k A\{b/x\}$.*

---

[13] `agda/reduction.agda` [14] `coq/axioms.v:DEquiv_Arrow_inj1,DEquiv_Arrow_inj2,DEquiv_Pi_inj1,DEquiv_Pi_inj2` [15] `coq/axioms.v:ineq_*` [16] `coq/ctx.v:DTyping_SubG` [17] `coq/subst.v:DTyping_subst`

Typing judgements themselves ensure the well-formedness of their components; in particular, if a term type checks, then its type can be typed at the same level. Because our type system includes the non–syntax-directed rule T-Conv, the proof of this lemma depends on several inversion lemmas, omitted here.

▶ **Lemma 13** (Regularity).[18] *If* $\Delta; \Gamma \vdash a :^k A$ *then* $\vdash \Delta$ *and* $\Delta \vdash \Gamma$ *and* $\Delta; \Gamma \vdash A :^k \star$

Generalizing displaceability in an empty context, derivations can be displaced wholesale by also incrementing contexts, written Γ^{+i}, where $(\Gamma, x :^k A)^{+i} = \Gamma^{+i}, x :^{k+i} A^{+i}$.

▶ **Lemma 14** (Displaceability).[19] *If* $\Delta; \Gamma \vdash a :^k A$ *then* $\Delta; \Gamma^{+j} \vdash a^{+j} :^{j+k} A^{+j}$.

If we displace a context, the result might not be stronger because displacement may modify the types in the assumptions. In other words, it is *not* the case that $\Gamma \leq \Gamma^{+k}$.

4.3 Restriction

The key idea of stratification is that a judgement at level k is only allowed to depend on judgements at the same or lower levels. One way to observe this property is through a form of strengthening result, which allows variables from higher levels to be removed from the context and contexts to be truncated at any level. Formally, we define the *restriction* operation, written $\lceil \Gamma \rceil^k$, which filters out all assumptions from the context with level greater than k. A restricted context can be stronger since it could contain fewer assumptions.

▶ **Lemma 15** (Restriction).[20] *If* $\Delta \vdash \Gamma$ *then* $\Delta \vdash \lceil \Gamma \rceil^k$ *for any* k, *and if* $\Delta; \Gamma \vdash a :^k A$ *then* $\Delta; \lceil \Gamma \rceil^k \vdash a :^k A$.

▶ **Lemma 16** (Restriction subsumption).[21] $\Gamma \leq \lceil \Gamma \rceil^k$.

4.3.1 Restricted floating

Subsumption allows variables from one level to be made available to all higher levels using their current type. However, when we use this rule in a judgement, it doesn't change the context that is used to check the term. This can be restrictive — we can only substitute their assumptions with lower level derivations.

In some cases, we can raise the level of some assumptions in the context when we raise the level of the judgement without displacing their types or the rest of the context. For example, consider the derivable judgement $f :^j \Pi x :^i A.\, B, x :^i A \vdash f\ x :^j B$ where $i < j$. We could derive the same judgement at a higher level $k > j$ where we also raise the level of f to k. However, we can only raise the level of variables at the *same* level as the entire judgement. In our example, we can't raise x from its lower level i because then it would be invalid as an argument to f.

To prove this formally, we must work with judgements that don't have any assumptions above the current level by using the restriction operation to discard them. Next, to raise certain levels, we introduce a *floating* operation on contexts $\uparrow^k_j \Gamma$ that raises assumptions in Γ at level j to a higher level k without displacing their types.

▶ **Lemma 17** (Restricted Floating).[22] *If* $\Delta; \Gamma \vdash a :^j A$ *and* $j \leq k$ *then* $\Delta; \uparrow^k_j(\lceil \Gamma \rceil^j) \vdash a :^k A$.

The restricted floating lemma is required to prove cumulativity of judgements.

▶ **Lemma 18** (Cumulativity).[23] *If* $\Delta; \Gamma \vdash a :^j A$ *and* $j \leq k$ *then* $\Delta; \Gamma \vdash a :^k A$.

---

[18] coq/ctx.v:DCtx_DSig , coq/inversion.v:DTyping_DCtx , coq/ctx.v:DTyping_regularity
[19] coq/ctx.v:DTyping_incr
[20] coq/ctx.v:DSig_DCtx_DTyping_restriction
[21] coq/restrict.v:SubG_restrict
[22] coq/restrict.v:DTyping_float_restrict
[23] coq/restrict.v:DTyping_cumul

In the nondependent function case $\Delta; \Gamma \vdash \lambda x.\, b :^{j} A \to B$, where we want to derive the same judgement at level $k \geq j$, we get by inversion the premise $\Delta; \Gamma, x :^{j} A \vdash b :^{j} B$, while we need $\Delta; \Gamma, x :^{k} A \vdash b :^{k} B$. Restricted floating and weakening allows us to raise the level of b together with the single assumption x from level j to level k.

4.3.2 Type Safety

We can now show that this language satisfies the preservation (*i.e.* subject reduction) and progress lemmas with respect to call-by-name $\beta\delta$-reduction $\boxed{\Delta \vdash a \leadsto b}$; the full set of reduction rules can be found in Appendix B. For progress, values are type formers and abstractions.

▶ **Lemma 19** (Preservation).[24] *If $\Delta; \Gamma \vdash a :^{k} A$ and $\Delta \vdash a \leadsto a'$ then $\Delta; \Gamma \vdash a' :^{k} A$.*

▶ **Lemma 20** (Progress).[25] *If $\Delta; \varnothing \vdash a :^{k} A$ then a is a value or $\Delta \vdash a \leadsto b$ for some b.*

5 Prototype implementation

We have implemented a prototype type checker, which can be found at `https://github.com/plclub/StraTT` under the `impl/` directory, including a brief overview of the concrete syntax.[26] This implementation is based on `pi-forall` [45], a simple bidirectional type checker for a dependently-typed programming language.

For convenience, displacements and level annotations on dependent types can be omitted; the type checker then generates level metavariables in their stead. When checking a single global definition, constraints on level metavariables are collected, which form a set of integer inequalities on metavariables. An SMT solver checks that these inequalities are satisfiable by the naturals and finally provides a solution that minimizes the levels. Therefore, assuming the collected constraints are correct, if a single global definition has a solution, then a solution will always be found. However, we don't know if this holds for a *set* of global definitions, because the solution for a prior definition might affect whether a later definition that uses it is solveable. Determining what makes a solution "better" or "more general" to maximize the number of global definitions that can be solved is part of future work.

The implementation additionally features stratified datatypes, case expressions, and recursion, used to demonstrate the practicality of programming in StraTT. Restricting the datatypes to inductive types by checking strict positivity and termination of recursive functions is possible but orthogonal to stratification and thus out of scope for this work. The parameters and arguments of datatypes and their constructors respectively can be either floating (*i.e.* nondependent) or fixed (*i.e.* dependent), with their levels following rules analogous to those of nondependent and dependent functions. Additionally, datatypes and constructors can be displaced like constants, in that a displaced constructor only belongs to its datatype with the same displacement.

We include with our implementation a small core library,[27] and all the examples that appear in this paper have been checked by our implementation.[28] Appendix C examines three particular datatypes in depth: decidable types, propositional equality, and dependent pairs.

[24] `coq/typesafety.v:Reduce_Preservation`
[25] `coq/typesafety.v:progress`
[26] `impl/README.pi`
[27] `impl/pi/README.pi`
[28] `impl/pi/StraTT.pi`

6 Discussion

6.1 On consistency

The consistency of subStraTT tells us that the basic premise of using stratification in place of a universe hierarchy is sensible. However, it isn't necessarily an incremental step towards consistency of the full StraTT, as we've seen that directly adding floating functions to the logical relation doesn't work, and an entirely different approach may be needed after all.

One possible direction is to take inspiration from the syntactic metatheory, especially the Restricted Floating lemma, which is required specifically to show cumulativity of floating functions. Since cumulativity is exactly where the naïve addition of floating functions to the logical relation fails, the key may be to formulate this lemma semantically. This might require modifying the logical relation to involve contexts and to relate open terms instead.

Another possibility is based on the observation that due to cumulativity, floating functions appear to be parametric in its stratification level, at least starting from the smallest level at which it can be well typed. This suggests that some sort of relational model may help to interpret levels parametrically.

Nevertheless, we strongly believe that StraTT is indeed consistent. The Restriction lemma in particular intuitively tells us that nothing at higher levels could possibly be smuggled into a lower level to violate stratification. As a further confidence check, we have verified that three type-theoretic paradoxes possible in an ordinary type theory with type-in-type do *not* type check in our implementation. These paradoxes are Burali-Forti's paradox [8], Russell's paradox [38], and Hurkens' paradox [23], which all end up reaching a point where a higher-level term needs to fit into a lower-level position to proceed any further — exactly what stratification is designed to prevent. Appendix D examines these paradoxes in depth.

6.2 On useability

Useability comes down to the balance between practicality and expressivity. On the practicality side, our implementation demonstrates that if a definition is well typed, then its levels and displacements can be completely omitted and inferred, a workflow comparable to Coq or Lean. Additionally, since constants are displaced by only a single displacement, StraTT doesn't exhibit the same kind of exponential blowup in levels and type checking time that can occur when using universe-polymorphic definitions in Coq or Lean, which need to abstract over and instantiate over all implicit levels involved. This behaviour is demonstrated by the concrete, though artificial, examples in Appendix E, whose corresponding StraTT definition checks just fine.[29] However, if a definition is *not* well typed, debugging it may involve wading through constraints between generated level metavariables in situations normally having nothing to do with universe levels, since stratification now involves levels everywhere, in particular when using dependent function types.

On the expressivity side, the displacement system of StraTT falls somewhere between level monomorphism and prenex level polymorphism; in some scenarios, it works just as well as polymorphism. For instance, to type check Hurkens' paradox as far as StraTT can, the Coq formulation of the paradox without type-in-type requires turning on universe polymorphism, and the Agda formulation of the paradox without type-in-type requires definitions polymorphic over at least three universe levels. In general, displacement seems particularly suited for our stratified system, since level annotations only appear on dependent

[29] `impl/pi/Blowup.pi`

function domains, not on universes. For example, the type $\Pi X :^0 \star.\, (X \to \star) \to \star$ only has one level, while the corresponding most general Agda type `(X : Set ℓ₁) → (X → Set ℓ₂) → Set ℓ₃` has three and would fare poorly with displacement.

However, in other scenarios, the expressivity of level polymorphism over multiple level variables is truly needed. For instance, merely having a type constructor with both a dependent domain and a nondependent domain interacts poorly with cumulativity. Suppose we had some type constructor $\mathsf{T} :^1 \Pi x :^0 X.\, Y \to \star$ and a function over elements of this type $\mathsf{f} :^1 \Pi x :^0 X.\, \Pi y :^0 Y.\, \mathsf{T}\ x\ y \to Z$. By cumulativity, if y has level 2, $\mathsf{T}\ x\ y$ is still well typed by cumulativity at level 2, but f can no longer be applied to it, since the level of y is now too high. We would like the second argument of f to float along with T, but this isn't possible since it's depended upon. Having the level of the second argument be polymorphic (subject to the expected constraints) would resolve this issue.

6.3 Related work

StraTT is directly inspired from Leivant's stratified polymorphism [26, 27, 14], which developed from Statman's ramified polymorphic typed λ-calculus [41]. Stratified System F, a slight modification of the original system, has since been used to demonstrate a normalization proof technique using hereditary substitution [18], which in turn has been mechanized in Coq as a case study for the Equations package [29]. More recently, an interpreter of an intrinsically-typed Stratified System F has been mechanized in Agda by Thiemann and Weidner [43], where stratification levels are interpreted as Agda's universe levels. Similarly, Hubers and Morris' Stratified R_ω, a stratified System F_ω with row types, has been mechanized in Agda as well [22]. Meanwhile, our system of level displacement comes from McBride's crude-but-effective stratification [33, 32], specializing the displacement algebra (in the sense of Favonia, Angiuli, and Mullanix [21]) to the naturals.

7 Conclusion

In this work, we have introduced Stratified Type Theory, a departure from a decades-old tradition of universe hierarchies without, we believe, succumbing to the threat of logical inconsistency. By stratifying dependent function types, we obstruct the usual avenues by which paradoxes manifest their inconsistencies; and by separately introducing floating nondependent function types, we recover some of the expressivity lost under the strict rule of stratification. Although proving logical consistency for the full StraTT remains future work, we *have* proven it for the subsystem subStraTT, and we have provided supporting evidence by showing how well-known type-theoretic paradoxes fail.

Towards demonstrating that StraTT isn't a mere theoretical exercise and, if consistent, is a viable basis for theorem proving and dependently-typed programming, we have implemented a prototype type checker for the language augmented with datatypes, along with a small core library. The implementation also features inference for level annotations and displacements, allowing the user to omit them entirely. We leave formally ensuring that our rules for datatypes don't violate existing metatheoretical properties as future work as well.

Given the various useability tradeoffs discussed, as well as the incomplete status of its consistency, we don't see any particularly compelling reason for existing proof assistants to adopt a system based on StraTT, but we don't anticipate any particular showstoppers, either, and believe it suitable for further improvement and iteration. Ultimately, we hope that StraTT demonstrates the feasibility of a renewed alternative to how type universes are handled, and opens up fresh avenues in the design space of type theories for proof assistants.

References

1. Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of Conversion for Type Theory in Type Theory. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158111.
2. Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, Pierre-Marie Pédrot, and Loïc Pujet. Martin-Löf à la Coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2024, page 230–245, 2024. doi:10.1145/3636501.3636951.
3. Brian Aydemir and Stephanie Weirich. LNgen: Tool Support for Locally Nameless Representations. Technical report, University of Pennsylvania, June 2010. doi:20.500.14332/7902.
4. Aydemir, Brian and Charguéraud, Arthur and Pierce, Benjamin C. and Pollack, Randy and Weirich, Stephanie. Engineering formal metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, page 3–15, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1328438.1328443.
5. Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. North-Holland, 1984.
6. Frédéric Blanqui. Inductive types in the Calculus of Algebraic Constructions. In *Typed Lambda Calculi and Applications, 6th International Conference, TLCA 2003*, volume 2701 of *LNCS*, Valencia, Spain, June 2003. URL: https://inria.hal.science/inria-00105617.
7. Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.
8. Cesare Burali–Forti. Una questione sui numeri transfiniti. *Rendiconti del Circolo matematico di Palermo*, 11, 1897.
9. Andrew V. Clifton. *Arend — Proof-assistant assisted pedagogy*. Master's thesis, California State University, Fresno, California, USA, 2015. URL: https://staffwww.fullcoll.edu/aclifton/files/arend-report.pdf.
10. The Coq Development Team. The Coq Proof Assistant, January 2022. URL: https://coq.github.io/doc/v8.15/refman, doi:10.5281/zenodo.5846982.
11. Thierry Coquand. The paradox of trees in type theory. *BIT Numerical Mathematics*, 32:10–14, March 1992. doi:10.1007/BF01995104.
12. Thierry Coquand. A new paradox in type theory. In *Studies in Logic and the Foundations of Mathematics*, volume 134, pages 555–570. Elsevier, 1995. doi:10.1016/S0049-237X(06)80062-5.
13. Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88*, volume 417, pages 50–66. Springer Berlin Heidelberg. URL: http://link.springer.com/10.1007/3-540-52335-9_47, doi:10.1007/3-540-52335-9_47.
14. Normal Danner and Daniel Leivant. Stratified polymorphism and primitive recursion. *Mathematical Structures in Computer Science*, 9(4):507–522, 1999. doi:10.1017/S0960129599002868.
15. Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In *International Conference on Automated Deduction*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388, August 2015. doi:10.1007/978-3-319-21401-6_26.
16. Dominique Devriese. [Agda] Simple contradiction from type-in-type, March 2013. URL: https://lists.chalmers.se/pipermail/agda/2013/005164.html.
17. Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. 65(2):525–549, June 2000. doi:10.2307/2586554.
18. Harley Eades III and Aaron Stump. Hereditary substitution for stratified System F. In *International Workshop on Proof Search in Type Theories*, 2010. URL: https://hde.design/includes/pubs/PSTT10.pdf.
19. Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD dissertation, Université Paris VII, 1972.
20. Martin Hofmann and Thomas Streicher. The groupoid model refutes uniqueness of identity proofs. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science (LICS 1994)*, pages 208–212. IEEE Computer Society Press, July 1994.

21 Kuen-Bang Hou (Favonia), Carlo Angiuli, and Reed Mullanix. An Order-Theoretic Analysis of Universe Polymorphism. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:10.1145/3571250.

22 Alex Hubers and J. Garrett Morris. Generic Programming with Extensible Data Types: Or, Making Ad Hoc Extensible Data Types Less Ad Hoc. *Proceedings of the ACM on Programming Languages*, 7(ICFP):356–384, Aug 2023. doi:10.1145/3607843.

23 Antonius J. C. Hurkens. A simplification of Girard's paradox. In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Typed Lambda Calculi and Applications*, pages 266–278, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

24 András Kovács. Generalized Universe Hierarchies and First-Class Universe Levels. In Florin Manea and Alex Simpson, editors, *30th EACSL Annual Conference on Computer Science Logic (CSL 2022)*, volume 216 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:17, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2022/15748, doi:10.4230/LIPIcs.CSL.2022.28.

25 Gottfried Wilhelm Leibniz. Discours de métaphysique, 1686.

26 Daniel Leivant. Stratified polymorphism. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 39–47, 1989. doi:10.1109/LICS.1989.39157.

27 Daniel Leivant. Finitely stratified polymorphism. *Information and Computation*, 93(1):93–113, 1991. Selections from 1989 IEEE Symposium on Logic in Computer Science. doi:10.1016/0890-5401(91)90053-5.

28 Yiyun Liu. Mechanized consistency proof for MLTT, 2024. Proof pearl under submission. URL: https://github.com/yiyunliu/mltt-consistency/.

29 Cyprien Mangin and Matthieu Sozeau. Equations for Hereditary Substitution in Leivant's Predicative System F: A Case Study. In *Tenth International Workshop on Logical Frameworks and Meta Languages: Theory and Practice*, volume 185 of *EPTCS*, Berlin, Germany, August 2015. URL: https://hal.inria.fr/hal-01248807, doi:10.4204/EPTCS.185.5.

30 Per Martin-Löf. A theory of types, 1971.

31 Per Martin-Löf. An intuitionistic theory of types, 1972.

32 Conor McBride. Crude but Effective Stratification, 2002. URL: https://personal.cis.strath.ac.uk/conor.mcbride/Crude.pdf.

33 Conor McBride. Crude but Effective Stratification, 2011. URL: https://mazzo.li/epilogue/index.html%3Fp=857&cpage=1.html.

34 Conor McBride. Datatypes of Datatypes, July 2015. URL: https://www.cs.ox.ac.uk/projects/utgp/school/conor.pdf.

35 Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers University of Technology and Göteborg University, Göteborg, Sweden, 2007. URL: https://research.chalmers.se/en/publication/46311.

36 John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, pages 408–425, Berlin, Heidelberg, 1974. Springer Berlin Heidelberg.

37 John C. Reynolds. Polymorphism is not set-theoretic. In Gilles Kahn, David B. MacQueen, and Gordon Plotkin, editors, *Semantics of Data Types*, pages 145–156, Berlin, Heidelberg, 1984. Springer Berlin Heidelberg. doi:10.1007/3-540-13346-1_7.

38 Bertrand Russell. *The Principles of Mathematics.* Cambridge University Press, 1903.

39 Peter Sewell, Franceso Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1):71–122, 2010. doi:10.1017/S0956796809990293.

40 Vilhelm Sjöberg. Why must inductive types be strictly positive?, April 2015. URL: https://vilhelms.github.io/posts/why-must-inductive-types-be-strictly-positive/.

41 Richard Statman. Number theoretic functions computable by polymorphic programs. In *22nd Annual Symposium on Foundations of Computer Science (SFCS 1981)*, pages 279–282, 1981. doi:10.1109/SFCS.1981.24.

42 Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent Types and Multi-Monadic Effects in F*. In *Principles of Programming Languages*, pages 256–270, January 2016. doi:10.1145/2837614.2837655.

43 Peter Thiemann and Marius Weidner. Towards Tagless Interpretation of Stratified System F. In Youyou Cong and Pierre-Evariste Dagand, editors, *TyDe 2023: Proceedings of the 8th ACM SIGPLAN International Workshop on Type-Driven Development*, 2023. URL: https://icfp23.sigplan.org/details/tyde-2023/12/.

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

45 Stephanie Weirich. Implementing Dependent Types in pi-forall, 2023. URL: https://arxiv.org/abs/2207.02129, doi:10.48550/arXiv.2207.02129.

A Well-formedness and equality

$\boxed{\vdash \Delta}$ *(Signature formation)*

$$\textsc{D-Empty}\quad \frac{}{\vdash \varnothing} \qquad\qquad \textsc{D-Cons}\quad \frac{\begin{array}{c}\vdash \Delta \qquad \Delta; \varnothing \vdash A :^k \star \\ \Delta; \varnothing \vdash a :^k A \\ x \notin \mathsf{dom}\, \Delta\end{array}}{\vdash \Delta, x :^k A \coloneqq a}$$

$\boxed{\Delta \vdash \Gamma}$ *(Context formation)*

$$\textsc{DG-Empty}\quad \frac{\vdash \Delta}{\Delta \vdash \varnothing} \qquad\qquad \textsc{DG-Cons}\quad \frac{\begin{array}{c}\Delta \vdash \Gamma \qquad \Delta; \Gamma \vdash A :^k \star \\ x \notin \mathsf{dom}\, \Gamma \\ x \notin \mathsf{dom}\, \Delta\end{array}}{\Delta \vdash \Gamma, x :^k A}$$

$\boxed{\Delta \vdash a \equiv b}$ *(Definitional equality)*

$$\textsc{DE-Refl}\quad \frac{}{\Delta \vdash a \equiv a} \qquad \textsc{DE-Sym}\quad \frac{\Delta \vdash b \equiv a}{\Delta \vdash a \equiv b} \qquad \textsc{DE-Trans}\quad \frac{\Delta \vdash a \equiv b \qquad \Delta \vdash b \equiv c}{\Delta \vdash a \equiv c} \qquad \textsc{DE-Beta}\quad \frac{}{\Delta \vdash (\lambda x.\, b)\ a \equiv b\{a/x\}}$$

$$\textsc{DE-Delta}\quad \frac{x :^k A \coloneqq a \in \Delta}{\Delta \vdash x^i \equiv a^{+i}} \qquad \textsc{DE-Arrow}\quad \frac{\begin{array}{c}\Delta \vdash A \equiv A' \\ \Delta \vdash B \equiv B'\end{array}}{\Delta \vdash A \to B \equiv A' \to B'} \qquad \textsc{DE-Pi}\quad \frac{\begin{array}{c}\Delta \vdash A \equiv A' \\ \Delta \vdash B \equiv B'\end{array}}{\Delta \vdash \Pi x :^k A.\, B \equiv \Pi x :^k A'.\, B'}$$

$$\textsc{DE-Abs}\quad \frac{\Delta \vdash b \equiv b'}{\Delta \vdash \lambda x.\, b \equiv \lambda x.\, b'} \qquad \textsc{DE-App}\quad \frac{\Delta \vdash a \equiv a' \qquad \Delta \vdash b \equiv b'}{\Delta \vdash b\ a \equiv b'\ a'} \qquad \textsc{DE-Absurd}\quad \frac{\Delta \vdash b \equiv b'}{\Delta \vdash \mathsf{absurd}(b) \equiv \mathsf{absurd}(b')}$$

Figure 3 Signature formation, context formation, and definitional equality rules

B Reduction

$\boxed{\Delta \vdash a \rightsquigarrow b}$ *(Single-step reduction)*

$$\textsc{R-Beta}\quad \frac{}{\Delta \vdash (\lambda x.\, b)\ a \rightsquigarrow b\{a/x\}}$$

$$\textsc{R-Delta}\quad \frac{x :^{k} A \coloneqq a \in \Delta}{\Delta \vdash x^{i} \rightsquigarrow a^{+i}}$$

$$\textsc{R-App}\quad \frac{\Delta \vdash b \rightsquigarrow b'}{\Delta \vdash b\ a \rightsquigarrow b'\ a}$$

$$\textsc{R-Absurd}\quad \frac{\Delta \vdash b \rightsquigarrow b'}{\Delta \vdash \mathsf{absurd}(b) \rightsquigarrow \mathsf{absurd}(b')}$$

$\boxed{\Delta \vdash a \rightsquigarrow^{*} b}$ *(Multi-step reduction)*

$$\textsc{W-Refl}\quad \frac{}{\Delta \vdash a \rightsquigarrow^{*} a}$$

$$\textsc{W-Trans}\quad \frac{\Delta \vdash a \rightsquigarrow b \qquad \Delta \vdash b \rightsquigarrow^{*} c}{\Delta \vdash a \rightsquigarrow^{*} c}$$

Figure 4 Call-by-name reduction

C Datatypes

C.1 Decidable types

Revisiting an example from Section 3, we can define Dec as a datatype.

$\textbf{data}\ \mathsf{Dec}\ (X : \star) :^{0} \star\ \textbf{where}$
$\quad \mathsf{Yes} :^{0} X \to \mathsf{Dec}\ X$
$\quad \mathsf{No} :^{0} \mathsf{neg}\ X \to \mathsf{Dec}\ X$

The lack of annotation on the parameter indicates that it's a floating domain, so that $\lambda X.\, \mathsf{Dec}\ X$ can be assigned type $\star \to \star$ at level 0. Datatypes and their constructors, like variables and constants, are cumulative, so the aforementioned type assignment is valid at any level above 0 as well. When destructing a datatype, the constructor arguments of each branch are typed such that the constructor would have the same level as the level of the scrutinee. Consider the following proof that decidability of a type implies its double negation elimination, which requires inspecting the decision.

$\mathsf{decDNE} :^{1} \Pi X :^{0} \star.\, \mathsf{Dec}\ X \to \mathsf{neg}\ (\mathsf{neg}\ X) \to X$
$\mathsf{decDNE}\ X\ \mathit{dec}\ \mathit{nn} \coloneqq \textbf{case}\ \mathit{dec}\ \textbf{of}$
$\quad \mathsf{Yes}\ y \Rightarrow y$
$\quad \mathsf{No}\ x \Rightarrow \mathsf{absurd}(\mathit{nn}\ x)$

By the level annotation on the function, we know that *dec* and *nn* both have level 1. Then in the branches, the patterns Yes y and No x must also be typed at level 1, so that y has type X and x has type neg X both at level 1.

C.2 Propositional equality

Datatypes and their constructors, like constants, can be displaced as well, uniformly raising the levels of their types. We again revisit an example from Section 3 and now define a propositional equality as a datatype with a single reflexivity constructor.

$$\begin{array}{l} \textbf{data}\ \mathsf{Eq}\ (X :^0 \star) :^1 X \to X \to \star\ \textbf{where} \\ \quad \mathsf{Refl} :^1 \Pi x{:}^0 X.\, Eq\ X\ x\ x \end{array}$$

This time, the parameter has a level annotation indicating that it's fixed at 0, while its indices are floating. Displacing Eq by 1 would then raise the fixed parameter level to 1, while the levels of Eq^1 itself and its floating indices always match but can be 2 or higher by cumulativity. Its sole constructor would be Refl^1 containing a single argument of type X at level 1. Displacement is needed to state and prove propositions about equalities between equalities, such as the uniqueness of equality proofs:[30]

$$\begin{array}{l} \mathsf{UIP} :^2 \Pi X{:}^0 \star.\, \Pi x{:}^0 X.\, \Pi p{:}^1 \mathsf{Eq}\ X\ x\ x.\, \mathsf{Eq}^1\ (\mathsf{Eq}\ X\ x\ x)\ p\ (\mathsf{Refl}\ x) \\ \mathsf{UIP}\ X\ x\ p \coloneqq \textbf{case}\ p\ \textbf{of}\ \mathsf{Refl}\ x \Rightarrow \mathsf{Refl}^1\ (\mathsf{Refl}\ x) \end{array}$$

C.3 Dependent pairs

Because there are two different function types, there are also two different ways to define dependent pairs. Using a floating function type for the second component's type results in pairs whose first and second projections can be defined as usual, while using the stratified dependent function type results in pairs whose second projection can't be defined in terms of the first. We first take a look at the former.

$$\begin{array}{l} \textbf{data}\ \mathsf{NPair}\ (X :^0 \star)\ (P : X \to \star) :^1 \star\ \textbf{where} \\ \quad \mathsf{MkPair} :^1 \Pi x{:}^0 X.\, P\ x \to \mathsf{NPair}\ X\ P \\ \mathsf{nfst} :^1 \Pi X{:}^0 \star.\, \Pi P{:}^0 X \to \star.\, \mathsf{NPair}\ X\ P \to X \\ \mathsf{nfst}\ X\ P\ p \coloneqq \textbf{case}\ p\ \textbf{of}\ \mathsf{MkPair}\ x\ y \Rightarrow x \\ \mathsf{nsnd} :^2 \Pi X{:}^0 \star.\, \Pi P{:}^0 X \to \star.\, \Pi p{:}^1 \mathsf{NPair}\ X\ P.\, P\ (\mathsf{nfst}\ X\ P\ p) \\ \mathsf{nsnd}\ X\ P\ p \coloneqq \textbf{case}\ p\ \textbf{of}\ \mathsf{MkPair}\ x\ y \Rightarrow y \end{array}$$

Due to stratification, the projections need to be defined at level 1 and 2 respectively to accommodate dependently quantifying over the parameters at level 0 and the pair at level 1. Even so, the second projection is well typed, since P can be used at level 2 by subsumption to be applied to the first projection at level 2 also by subsumption in the return type of the second projection.

As the two function types are distinct, we do need both varieties of dependent pairs. In particular, with the above pairs alone, we aren't able to type check a universe of propositions $\mathsf{NPair}\ \star\ \mathsf{isProp}$, as the predicate has type $\Pi X{:}^0 \star.\, \star$ at level 1.

$$\begin{array}{l} \textbf{data}\ \mathsf{DPair}\ (X :^0 \star)\ (P : \Pi x{:}^0 X.\, \star) :^1 \star\ \textbf{where} \\ \quad \mathsf{MkPair} :^1 \Pi x{:}^0 X.\, P\ x \to \mathsf{DPair}\ X\ P \\ \mathsf{dfst} :^2 \Pi X{:}^0 \star.\, \Pi P{:}^1 (\Pi x{:}^0 X.\, \star).\, \mathsf{DPair}\ X\ P \to X \end{array}$$

[30] The provability of this principle, also known as UIP [20], is more a consequence of the quirks of unification in `pi-forall` than an intentional intensional design.

$\mathsf{dfst}\ X\ P\ p \coloneqq \textbf{case}\ p\ \textbf{of}\ \mathsf{MkPair}\ x\ y \Rightarrow x$
$\mathsf{dsnd} :^2 \Pi X :^0 \star.\, \Pi P :^1 (\Pi x :^0 X.\, \star).\, \Pi p :^1 \mathsf{DPair}\ X\ P.$
$\quad \textbf{case}\ p\ \textbf{of}\ \mathsf{MkPair}\ x\ y \Rightarrow P\ x$
$\mathsf{dsnd}\ X\ P\ p \coloneqq \textbf{case}\ p\ \textbf{of}\ \mathsf{MkPair}\ x\ y \Rightarrow y$

In the second variant of dependent pairs where P is a stratified dependent function type, the domain of P is fixed to level 0, so in the type in dsnd, it can't be applied to the first projection, but it can still be applied to the first component by matching on the pair. Now we're able to type check DPair $\star$ isProp.

In both cases, the first component has a fixed level, while the second component is floating, so using a predicate at a higher level results in a pair type at a higher level by subsumption. Consider the predicate isSet, which has type $\Pi X :^0 \star.\, \star$ at level 2: the universe of sets DPair $\star$ isSet is also well typed at level 2.

Unfortunately, the first projection dfst can no longer be used on an element of this pair, since the predicate is now at level 2, nor can its displacement dfst^1, since that would displace the level of the first component as well. Without proper level polymorphism, which would allow keeping the first argument's level fixed while setting the second argument's level to 2, we're forced to write a whole new first projection function.

In general, this limitation occurs whenever a datatype contains both dependent and nondependent parameters. Nevertheless, in the case of the pair type, the flexibility of a nondependent second component type is still preferable to a dependent one that fixes its level, since there would need to be entirely separate datatype definitions for different combinations of first and second component levels, *i.e.* one with levels 0 and 1 (as in the case of isProp), one with levels 0 and 2 (as in the case of isSet), and so on.

D Paradoxes

D.1 Burali-Forti's paradox

Burali-Forti's paradox [8] in set theory concerns the simultaneous well-foundedness and non–well-foundedness of an ordinal. In type theory, we instead consider a particular datatype U due to Coquand [11],[31,32] along with a well-foundedness predicate for U.

$\textbf{data}\ \mathsf{U} :^1 \star\ \textbf{where}$
$\quad \mathsf{MkU} :^1 \Pi X :^0 \star.\, (X \to \mathsf{U}) \to \mathsf{U}$
$\textbf{data}\ \mathsf{WF} :^2 \mathsf{U} \to \star\ \textbf{where}$
$\quad \mathsf{MkWF} :^2 \Pi X :^0 \star.\, \Pi f :^1 X \to \mathsf{U}.\, (\Pi x :^1 X.\, \mathsf{WF}\ (f\ x)) \to \mathsf{WF}\ (\mathsf{MkU}\ X\ f)$

Note that both of these definitions are strictly positive, so we aren't using any tricks relying on negative datatypes. It's easy to show that all U are well founded.

$\mathsf{wf} :^2 \Pi u :^1 \mathsf{U}.\, \mathsf{WF}\ u$
$\mathsf{wf}\ u \coloneqq \textbf{case}\ u\ \textbf{of}$
$\quad \mathsf{MkU}\ X\ f \Rightarrow \mathsf{MkWF}\ X\ f\ (\lambda x.\, \mathsf{wf}\ (f\ x))$

[31] Our thanks to Stephen Dolan for detailing to us this example. [32] `impl/pi/WFU.pi`

The usual paradox, with type-in-type and without stratification, constructs a U that is provably *not* well founded.

$$\begin{aligned}
&\mathsf{loop} :^1 \mathsf{U}\\
&\mathsf{loop} \coloneqq \mathsf{MkU}\ \underline{\mathsf{U}}\ (\lambda u.\, u)\\
&\mathsf{nwfLoop} :^2 \mathsf{WF}\ \mathsf{loop} \to \bot\\
&\mathsf{nwfLoop}\ \mathit{wfLoop} \coloneqq \mathbf{case}\ \mathit{wfLoop}\ \mathbf{of}\\
&\quad \mathsf{MkWF}\ X\ f\ h \Rightarrow \mathsf{nwfLoop}\ (h\ \mathsf{loop})
\end{aligned}$$

In the branch of nwfLoop, by pattern matching on the type of the scrutinee, X is bound to U and f to $\lambda u.\, u$, so h loop correctly has type WF loop. Note that this definition would also pass the usual structural termination check, since the recursive call is done on a subargument from h. Then nwfLoop (wf loop) is an inhabitant of the empty type.

With stratification, U with level 1 is too large to fit into the type argument of MkU, which demands level 0, so loop can't be constructed in the first place. This is also why the level of a datatype can't be strictly lower than that of its constructors, despite such a design not violating the regularity lemma for constructors.

D.2 Russell's paradox

The U above was originally used by Coquand [11] to express a variant of Russell's paradox [38].[33,34] First, a U is said to be regular if it's provably inequal to its subarguments; this represents a set which doesn't contain itself.

$$\begin{aligned}
&\mathsf{regular} :^1 \mathsf{U} \to \star\\
&\mathsf{regular}\ u \coloneqq \mathbf{case}\ u\ \mathbf{of}\\
&\quad \mathsf{MkU}\ X\ f \Rightarrow \Pi x{:}^0 X.\, (f\ x = \mathsf{MkU}\ X\ f) \to \bot
\end{aligned}$$

The trick is to define a U that is both regular and nonregular. Normally, with type-in-type, this would be one that represents the set of all regular sets.

$$\begin{aligned}
&\mathsf{R} :^3 \mathsf{U}^2\\
&\mathsf{R} \coloneqq \mathsf{MkU}^2\ (\mathsf{NPair}^1\ \mathsf{U}\ \mathsf{regular})\ \underline{(\mathsf{nfst}^1\ \mathsf{U}\ \mathsf{regular})}
\end{aligned}$$

Stratification once again prevents R from type checking, since the pair projection returns a U and not a U^2. The type contained in the pair can't be displaced to U^2 either, since that would make the pair's level too large to fit inside MkU^2.

D.3 Hurkens' paradox

Although we've seen that stratification thwarts the paradoxes above, they leverage the properties of datatypes and recursive functions, which we haven't formalized. Here, we'll turn to the failure of Hurkens' paradox [23] as further evidence of consistency, which in contrast can be formulated in pure StraTT without datatypes. Below is the paradox in Coq without universe checking.

[33] An Agda implementation can be found at `https://github.com/agda/agda/blob/master/test/Succeed/Russell.agda` [16].
[34] `impl/pi/Russell.pi`

```
Require Import Coq.Unicode.Utf8_core.
Unset Universe Checking.
Definition P (X : Type) : Type := X → Type.
Definition U : Type :=
  ∀ (X : Type), (P (P X) → X) → P (P X).
Definition tau (t : P (P U)) : U :=
  λ X f p, t (λ s, p (f (s X f))).
Definition sig (s : U) : P (P U) := s U tau.
Definition Delta (y : U) : Type :=
  (∀ (p : P U), sig y p → p (tau (sig y))) → False.
Definition Omega : U :=
  tau (λ p, ∀ (x : U), sig x p → p x).
Definition M (x : U) (s : sig x Delta) : Delta x :=
  λ d, d Delta s (λ p, d (λ y, p (tau (sig y)))).
Definition D := ∀ p, (∀ x, sig x p → p x) → p Omega.
Definition R : D :=
  λ p d, d Omega (λ y, d (tau (sig y))).
Definition L (d : D) : False :=
  d Delta M (λ p, d (λ y, p (tau (sig y)))).
Definition false : False := L R.
```

If we replace unsetting universe checking with

```
Set Universe Polymorphism.
```

then the definitions check up to M. Similarly, in Agda, we can get the paradox to check up to M by using explicit universe polymorphism.

```
{-# OPTIONS --cumulativity #-}
open import Agda.Primitive

data ⊥ : Set where

U : ∀ ℓ ℓ₁ ℓ₂ → Set (lsuc (ℓ ⊔ ℓ₁ ⊔ ℓ₂))
U ℓ ℓ₁ ℓ₂ = ∀ (X : Set ℓ) → (((X → Set ℓ₁) → Set ℓ₂) → X) → ((X → Set ℓ₁) → Set ℓ₂)

τ : ∀ ℓ₁ ℓ₂ → ((U ℓ₁ ℓ₁ ℓ₂ → Set ℓ₁) → Set ℓ₂) → U ℓ₁ ℓ₁ ℓ₂
τ ℓ₁ ℓ₂ t = λ X f p → t (λ x → p (f (x X f)))

σ : ∀ ℓ₁ ℓ₂ → U (lsuc (ℓ₁ ⊔ ℓ₂)) ℓ₁ ℓ₂ → (U ℓ₁ ℓ₁ ℓ₂ → Set ℓ₁) → Set ℓ₂
σ ℓ₁ ℓ₂ s = s (U ℓ₁ ℓ₁ ℓ₂) (τ ℓ₁ ℓ₂)

Δ : ∀ {ℓ₁ ℓ₂} → U (lsuc (ℓ₁ ⊔ ℓ₂)) ℓ₁ ℓ₂ → Set (lsuc (ℓ₁ ⊔ ℓ₂))
Δ {ℓ₁} {ℓ₂} y = (∀ p → σ ℓ₁ ℓ₂ y p → p (τ ℓ₁ ℓ₂ (σ ℓ₁ ℓ₂ y))) → ⊥

Ω : ∀ {ℓ} → U ℓ ℓ (lsuc (lsuc ℓ))
Ω {ℓ} = τ ℓ (lsuc (lsuc ℓ)) (λ p → (∀ x → σ ℓ ℓ x p → p x))

M : ∀ {ℓ} x → σ (lsuc ℓ) ℓ x (Δ {ℓ} {ℓ}) → Δ {lsuc ℓ} {ℓ} x
M {ℓ} _ 2 3 = 3 Δ 2 (λ p → 3 (λ y → p (τ ℓ ℓ (σ ℓ ℓ y))))
```

```
R : ∀ {ℓ} p → (∀ x → σ ℓ (lsuc (lsuc ℓ)) x p → p x) → p Ω
R {ℓ} _ 𝟙 = {! 𝟙 (Ω {ℓ}) (λ x → 𝟙 (τ ℓ ℓ (σ ℓ ℓ x))) !}
-- Need Ω : U (lsuc (lsuc (lsuc ℓ))) ℓ (lsuc (lsuc ℓ))
-- Have Ω : U ℓ ℓ (lsuc (lsuc ℓ))

L : ∀ {ℓ} → (∀ p → (∀ x → σ ℓ (lsuc (lsuc ℓ)) x p → p x) → p Ω) → ⊥
L {ℓ} 𝟘 = {! 𝟘 (Δ {ℓ} {ℓ}) M (λ p → 𝟘 (λ y → p (τ ℓ ℓ ℓ (σ ℓ ℓ ℓ y)))) !}
-- Need Δ : U ℓ ℓ (lsuc (lsuc ℓ)) → Set ℓ
-- Have Δ : U (lsuc ℓ) ℓ ℓ → Set (lsuc ℓ)

false : ⊥
false = L {lzero} (R {lzero})
```

The corresponding StraTT code, too, checks up to M, as verified in the implementation[35] Displacement is sufficient to handle situations in which polymorphism was needed.

$$\mathsf{P} :^0 \star \to \star$$
$$\mathsf{P}\ X \coloneqq X \to \star$$
$$\mathsf{U} :^1 \star$$
$$\mathsf{U} \coloneqq \Pi X :^0 \star.\, (\mathsf{P}\ (\mathsf{P}\ X) \to X) \to \mathsf{P}\ (\mathsf{P}\ X)$$
$$\mathsf{tau} :^1 \mathsf{P}\ (\mathsf{P}\ \mathsf{U}) \to \mathsf{U}$$
$$\mathsf{tau}\ t\ X\ f\ p \coloneqq t\ (\lambda s.\, p\ (f\ (s\ X\ f)))$$
$$\mathsf{sig} :^2 \mathsf{U}^1 \to \mathsf{P}\ (\mathsf{P}\ \mathsf{U})$$
$$\mathsf{sig}\ s \coloneqq s\ \mathsf{U}\ \mathsf{tau}$$
$$\mathsf{Delta} :^2 \mathsf{P}\ \mathsf{U}^1$$
$$\mathsf{Delta}\ y \coloneqq (\Pi p :^1 \mathsf{P}\ \mathsf{U}.\, \mathsf{sig}\ y\ p \to p\ (\mathsf{tau}\ (\mathsf{sig}\ y))) \to \bot$$
$$\mathsf{Omega} :^3 \mathsf{U}$$
$$\mathsf{Omega} \coloneqq \mathsf{tau}\ (\lambda p.\, \Pi x :^2 \mathsf{U}^1.\, \mathsf{sig}\ x\ p \to p\ (\lambda X.\, x\ X))$$
$$\mathsf{M} :^4 \Pi x :^3 \mathsf{U}^2.\, \mathsf{sig}^1\ x\ \mathsf{Delta} \to \mathsf{Delta}^1\ x$$
$$\mathsf{M}\ x\ s\ d \coloneqq d\ \mathsf{Delta}\ s\ (\lambda p.\, d\ (\lambda y.\, p\ (\mathsf{tau}\ (\mathsf{sig}\ y))))$$
$$\mathsf{D} :^3 \star$$
$$\mathsf{D} \coloneqq \Pi p :^1 \mathsf{P}\ \mathsf{U}.\, (\Pi x :^1 \mathsf{U}.\, \mathsf{sig}\ x\ p \to p\ x) \to p\ \mathsf{Omega}$$

The next definition D doesn't type check, since sig takes a displaced U^1 and not a U. The type of x can't be displaced to fix this either, since p takes an undisplaced U and not a U^1. Being stuck trying to equate two different levels is reassuring, as conflating different universe levels is how we expect a paradox that exploits type-in-type to operate.

D.4 Reynolds' paradox

Our final example concerns the inconsistency of inductives which are positive but not *strictly* so together with an impredicative universe, as described by Coquand and Paulin-Mohring [13].[36,37] We consider such a nonstrictly-positive datatype A_0.

---

[35] `impl/pi/Hurkens.pi` (no annotations), `impl/pi/HurkensAnnot.pi` (all annotations) [36] A Coq implementation has been made by Sjöberg [40]. [37] `impl/pi/Reynolds.pi`

$$\textbf{data}\ \mathsf{A_0} :^0 \star\ \textbf{where}$$
$$\mathsf{A_0} :^0 ((\mathsf{A_0} \to \star) \to \star) \to \mathsf{A_0}$$

$\mathsf{A_0}$ has one constructor whose only argument has type $(\mathsf{A_0} \to \star) \to \star$. Note that we don't need to use its induction principle (*i.e.* recursion), merely the fact that there's an injection from the latter type to the former, and so can be seen as a type-theoretic formulation of Reynolds' paradox [37]; this has also been detailed by Coquand [12].

We can define an injection f from $\mathsf{A_0} \to \star$ to $\mathsf{A_0}$. Injectivity of both $\mathsf{MkA_0}$ and f are omitted below; they are a crucial part of the paradox, but are orthogonal to what fails to type check.

$$\mathsf{f} :^0 (\mathsf{A_0} \to \star) \to \mathsf{A_0}$$
$$\mathsf{f}\ x \coloneqq \mathsf{MkA_0}\ (\lambda z.\, z = x)$$

Now we are in a position to define a property P similar to regularity from Russell's paradox above, and an element of $\mathsf{A_0}$ that simultaneously does and doesn't satisfy P.

$$\mathsf{P} :^1 \mathsf{A_0} \to \star$$
$$\mathsf{P}\ x \coloneqq \mathsf{NPair}\ (\mathsf{A_0} \to \star)\ (\lambda P.\, \mathsf{Pair}\ (x = f\ P)\ (P\ x \to \bot))$$
$$\mathsf{a_0} :^1 \mathsf{A_0}$$
$$\mathsf{a_0} \coloneqq \mathsf{f}\ \mathsf{P}$$

The details are omitted, but the where the paradox fails to type check is in trying to construct an element of $\mathsf{P}\ \mathsf{a_0}$ using P itself as the first element of the pair. Its level is 1, which is too high for the dependent pair, which asks for a first component at level 0; displacing NPair will raise the level of P, which will again make it still too high.

Impredicativity is what normally makes this paradox go through, disallowing nonstrictly-positive inductives for consistency. As StraTT is predicative, this may permit us to have nonstrictly-positive datatypes consistently; precedents include Blanqui's Calculus of Algebraic Constructions [6, Section 7].

E Exponential universe polymorphism

E.1 Coq

```
Set Universe Polymorphism.
Time Definition T1 : Type := Type -> Type -> Type -> Type -> Type -> Type.
Time Definition T2 : Type := T1 -> T1 -> T1 -> T1 -> T1 -> T1.
Time Definition T3 : Type := T2 -> T2 -> T2 -> T2 -> T2 -> T2.
Time Definition T4 : Type := T3 -> T3 -> T3 -> T3 -> T3 -> T3.
Time Definition T5 : Type := T4 -> T4 -> T4 -> T4 -> T4 -> T4.
Time Definition T6 : Type := T5 -> T5 -> T5 -> T5 -> T5 -> T5.
Time Definition T7 : Type := T6 -> T6 -> T6 -> T6 -> T6 -> T6.
Time Definition T8 : Type := T7 -> T7 -> T7 -> T7 -> T7 -> T7.
```

E.2 Lean

```
def T1 : Type _ := Type _ → Type _ → Type _ → Type _ → Type _ → Type _
def T2 : Type _ := T1 → T1 → T1 → T1 → T1 → T1
```

```
def T3 : Type _ := T2 → T2 → T2 → T2 → T2 → T2
def T4 : Type _ := T3 → T3 → T3 → T3 → T3 → T3
def T5 : Type _ := T4 → T4 → T4 → T4 → T4 → T4
def T6 : Type _ := T5 → T5 → T5 → T5 → T5 → T5
def T7 : Type _ := T6 → T6 → T6 → T6 → T6 → T6
def T8 : Type _ := T7 → T7 → T7 → T7 → T7 → T7
```