

# Stratified Type Theory

Jonathan Chan

University of Pennsylvania  
Philadelphia, USA  
jcxz@seas.upenn.edu

Stephanie Weirich

University of Pennsylvania  
Philadelphia, USA  
sweirich@seas.upenn.edu

## Abstract

To exploit the expressivity of being able to refer to the type of types, such as for large elimination, dependent type systems will either employ a universe hierarchy or else contend with an inconsistent type-in-type rule. However, these are not the only possible options. Taking inspiration from 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 function types to some fixed level strictly lower than that of the overall type. Even in the presence of type-in-type, this restriction suffices to enforce consistency.

We explore the expressivity of several extensions atop this design. First, the subsystem *subStrATT* employs McBride’s crude-but-effective stratification (also known as displacement) as a simple form of level polymorphism where top-level definitions can be displaced uniformly to any higher level as needed, which is valid due to cumulativity and plays well with stratified judgements. Second, to recover some expressivity lost due to the restriction on dependent function domains, the full StrATT system includes a separate nondependent function type with *floating* domains, whose level instead matches that of the overall type. Finally, we have implemented a prototype type checker for StrATT extended with datatypes along with a small type checked core library.

While the subsystem can be shown to be consistent, showing consistency for the full system with floating nondependent functions remains an open problem. Nevertheless, we believe that the full system is also consistent and have mechanized a syntactic proof of subject reduction. Furthermore, we use our implementation to investigate various well-known type-theoretic type-in-type paradoxes. These examples all fail to type check in expected ways as evidence towards consistency.

## 1 Introduction

Ever since their introduction in Martin-Löf’s intuitionistic type theory (MLTT) [22], dependent type theories have included hierarchies of type universes in order to rectify the inconsistency of the type-in-type axiom. That is, rather than the universe  $\star$  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 [6], Agda [25], Lean [8], F\* [30], Arend [5], and soon Idris 2 [3]. For greater expressiveness, all of these (except for Idris 2) 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?

To begin our exploration, let us take a look back at a different mechanism for universe levels and revisit type polymorphism in System F [12, 26]. Recall the formation rule for polymorphic type quantification in System F, given below on the left. This rule is part of the judgement  $\Gamma \vdash A \text{ type}$ , which asserts that the type  $A$  is well formed in context  $\Gamma$ . The quantification in this rule is impredicative because the type  $\forall x. B$  itself can be substituted for  $x$  in  $B$ , and it quantifies over all types including itself.

$$\begin{array}{c} \text{F-IMPREDICATIVE} \\ \Gamma, x \text{ type} \vdash B \text{ type} \\ \hline \Gamma \vdash \forall x. B \text{ type} \end{array} \qquad \begin{array}{c} \text{F-STRATIFIED} \\ \Gamma, x \text{ type} \vdash B \text{ type } k \quad j < k \\ \hline \Gamma \vdash \forall x^j. B \text{ type } k \end{array}$$

Impredicativity has long been a troublemaker in the metatheory of System F, in particular the lack of a classical set-theoretic model [27]. To sidestep impredicativity, Leivant [18] introduced *Stratified System F*, which stratifies types into different levels by disallowing quantifying over types at the same level as the quantification itself. The formation rule for polymorphic types in this system is shown in the above rule on the right. This rule is part of the stratified type formation judgement, written  $\Gamma \vdash A \text{ type } k$ , where  $k$  is a stratification level.

To extend stratified polymorphism to dependent types, there are two ways to read this judgement. We could interpret  $\Gamma \vdash A \text{ type } k$  as a type  $A$  living in some stratified type universe  $\star_k$ ; the generalization would then correspond to a usual predicative type theory with a universe hierarchy where  $\star_j : \star_k$  when  $j < k$ . Alternatively, we could interpret the level  $k$  as a property of the *judgement* rather than part of a type universe, and reexpress the judgement as  $\Gamma \vdash A :^k \star$ .

Since dependent types can depend on terms, we might generalize the stratified type formation judgement to a stratified typing judgement  $\Gamma \vdash a :^k A$ , where variables  $x :^k A$  are also annotated with a level within the context  $\Gamma$ , but using a

type universe that doesn't have a level annotation. Guided by these principles, we introduce stratified dependent function types  $\Pi x :^k A. B$  which similarly quantify over types at strictly lower levels.

To enable code reuse, rather than level polymorphism, we employ *crude but effective stratification* by McBride [24]. Following Hou (Favonia) et al. [13], we refer to this as *displacement* to prevent confusion. Given some signature  $\Delta$  of global definitions, we are permitted to use any definition with its levels displaced upwards. In the context of StraTT, displacement enables functions with level-annotated types to be used with arguments at any higher level.

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 non-dependent function type with a *floating* domain. StraTT is cumulative, so all expressions inhabit the level at which they type check and at all higher levels. However, in a dependent function type, the level of the domain type is fixed even when the overall level of the type has been raised. In a floating, non-dependent type, level of the domain type floats to have the same level as the overall type.

In the absence of floating non-dependent functions, with only stratified dependent functions, logical consistency holds even with type-in-type, because the restriction on the domains of dependent functions prevents the kind of self-referential trickery that enables the usual paradoxes. However, we have not yet proven logical consistency with the addition of floating non-dependent functions. The covariant behaviour of the floating domain with respect to levels is unusual for function types, and is the primary barrier to semantic modelling. Even so, we have not found proof of inconsistency either, and our attempts lead us to believe that consistency *does* hold, making the system suitable as a foundation for theorem proving.

These features form the basis of our **Stratified Type Theory** (StraTT). Our contributions are as follows:

- We first define subStraTT, a subsystem of StraTT, which features only stratified dependent function types and displacement. We briefly sketch a proof of consistency, modelling type universes with an inductive-recursive definition in Agda.  $\hookrightarrow$  Section 2
- We then extend this subsystem to the full StraTT by adding non-dependent function types with floating domains, motivated through examples.  $\hookrightarrow$  Section 3
- We have used the Coq proof assistant to prove important syntactic metatheorems for StraTT, including subject reduction, which is nontrivial due to the level-annotated context.  $\hookrightarrow$  Section 4
- We have developed a prototype implementation of a type checker, extending the language to include datatypes. We use this implementation to demonstrate the effectiveness

of stratification and displacement in practical dependently-typed programming, as well as its shortcomings when compared to prenex universe polymorphism.  $\hookrightarrow$  Section 5

- As evidence towards logical consistency, we discuss how common type-theoretic paradoxes, namely Hurkens' paradox [15] and variants of Russell's paradox [28] and Burali-Fori's paradox [4], fail to type check. We briefly highlight the challenges in a consistency proof attempt.  $\hookrightarrow$  Section 6

Section 7 discusses related work and we conclude in Section 8. Our Agda model, Coq metatheory, and prototype implementation are available online at <https://github.com/plclub/StraTT>. Where lemmas and theorems are first introduced, we include a footnote indicating the corresponding source file and lemma name in the development.

## 2 A subsystem of Stratified Type Theory

In this section, we introduce subStraTT, a fragment of StraTT that does not include the separate non-dependent function types. As it's a subsystem, the main theorems of subject reduction and other lemmas in Section 4 proven for the full StraTT still hold.

The subsystem subStraTT is a cumulative, extrinsic type theory with types à la Russell, a single type universe, level-annotated dependent function types, an empty type, and definitions with level displacement. 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 [13].

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  $\Gamma$  and signature  $\Delta$ . A signature consists of global definitions  $x :^k A := a$ , where each constant  $x$  is definitionally equal to its definition  $a$ . A context consists of declarations  $x :^k A$  of variables  $x$ .

The type of the type universe  $\star$  is itself at any level; in the next section, we show how even with this rule, subStraTT can be proven consistent. Stratification occurs at dependent function types in rule **DT-PI**: one can only quantify over types at strictly smaller levels, and the domain type must be well typed at the same strictly smaller level. 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 of the strictly smaller level indicated by the function's type.

Rules **DT-BOTTOM** and **DT-ABSURD** are the uninhabited type and its eliminator, respectively. Although it should be consistent to eliminate a falsehood into any level, including

$a, b, A, B ::= \quad \text{(Syntax)}$ $  \star   x   x^i   \Pi x.^j A. B   \lambda x. b   b a   \perp   \text{absurd}(b)$	$\vdash \Delta \quad \Delta; \emptyset \vdash A :^k \star$ $\frac{\Delta; \emptyset \vdash a :^k A \quad x \notin \text{dom } \Delta}{\vdash \Delta, x :^k A := a}$ $\frac{\Delta \vdash \Gamma \quad \Delta; \Gamma \vdash A :^k \star \quad x \notin \text{dom } \Gamma \quad x \notin \text{dom } \Delta}{\Delta \vdash \Gamma, x :^k A}$
$\boxed{\Delta; \Gamma \vdash a :^k A} \quad \text{(Typing)}$	$\text{DT-PI} \quad \frac{\Delta; \Gamma \vdash A :^j \star \quad \Delta; \Gamma, x :^j A \vdash B :^k \star \quad j < k}{\Delta; \Gamma \vdash \Pi x.^j A. B :^k \star}$
$\text{DT-TYPE} \quad \frac{\Delta \vdash \Gamma}{\Delta; \Gamma \vdash \star :^k \star}$	$\text{DT-APP} \quad \frac{\Delta; \Gamma \vdash b :^k \Pi x.^j A. B \quad \Delta; \Gamma \vdash a :^j A \quad j < k}{\Delta; \Gamma \vdash b a :^k B\{a/x\}}$
$\text{DT-ABSTY} \quad \frac{\Delta; \Gamma \vdash A :^j \star \quad \Delta; \Gamma, x :^j A \vdash b :^k B \quad j < k}{\Delta; \Gamma \vdash \lambda x. b :^k \Pi x.^j A. B}$	$\text{DT-CONST} \quad \frac{x :^j A := a \in \Delta \quad \Delta \vdash \Gamma \quad \vdash \Delta \quad i + j \leq k}{\Delta; \Gamma \vdash x^i :^k A^{+i}}$
$\text{DT-VAR} \quad \frac{\Delta \vdash \Gamma \quad x :^j A \in \Gamma \quad j \leq k}{\Delta; \Gamma \vdash x :^k A}$	$\text{DT-BOTTOM} \quad \frac{\Delta \vdash \Gamma}{\Delta; \Gamma \vdash \perp :^k \star}$
$\text{DT-ABSURD} \quad \frac{\Delta; \Gamma \vdash A :^k \star \quad \Delta; \Gamma \vdash b :^k \perp}{\Delta; \Gamma \vdash \text{absurd}(b) :^k A}$	$\text{DT-CONV} \quad \frac{\Delta; \Gamma \vdash a :^k A \quad \Delta; \Gamma \vdash B :^k \star \quad \Delta \vdash A \equiv B}{\Delta; \Gamma \vdash a :^k B}$

Figure 1. Syntax and typing rules (subStraTT)

lower levels, we restrict it so that the premises have the same level as the eliminator so that we can prove [Regularity](#).

In rules [DT-VAR](#) and [DT-CONST](#), variables and constants at level  $j$  can be used at any larger level  $k$ . This permits the following admissible cumulativity rule<sup>1</sup>, analogous to having a cumulative universe hierarchy, allowing instead an entire derivation to be used at a higher level.

$$\frac{\Delta; \Gamma \vdash a :^j A \quad j \leq k}{\Delta; \Gamma \vdash a :^k A}$$

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  $(x^j)^{+i} = x^{i+j}$ .

The key formation rules for signatures  $\boxed{\Gamma \Delta}$  and contexts  $\boxed{\Delta \vdash \Gamma}$  are given below.

<sup>1</sup>[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  $\beta\eta$ -equivalence for functions and  $\delta$ -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  $\beta$ -,  $\eta$ -, and  $\delta$ -equivalence; the remaining rules can be found in [Appendix A](#).

$$\frac{x :^k A := a \in \Delta}{\Delta \vdash (\lambda x. b) a \equiv b\{a/x\}} \quad \frac{}{\Delta \vdash \lambda x. b x \equiv b} \quad \frac{}{\Delta \vdash x^i \equiv a^{+i}}$$

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

**Lemma 2.1** (Displaceability (empty context)).<sup>2</sup> If  $\Delta; \emptyset \vdash a :^k A$  then  $\Delta; \emptyset \vdash a^{+i} :^{i+k} A^{+i}$ .

With  $x :^k A := a$  in the signature,  $x^i$  is definitionally equal to  $a^{+i}$ . Thus, this lemma justifies rule [DT-CONST](#), which gives such displaced constants  $x^i$  the type  $A^{+i}$ .

## 2.1 Consistency proof sketch

We can show that subStraTT is a consistent type theory, *i.e.* that not all types are inhabited. In this section, we sketch a model for subStraTT in Agda through the framework of *categories with families* [10], focussing on how types are modelled.

Inspired by Kovács [16], we use induction–recursion to model universes at each level, relying on the well-foundedness of levels to ensure their well-definedness. The elements of the inductive definition represent codes of the types of subStraTT, while the recursive function interprets these codes as types in Agda. Consistency follows from the interpretation of the empty type in subStraTT as an empty type in Agda, and so is relative to the consistency of Agda.

Because the interesting part of subStraTT when considering consistency is the presence of type-in-type, here we only include the model for universes, and omit constants and displacement, as well as the interpretation from subStraTT into the model. We have made the Agda files containing the whole model available at <https://github.com/plclub/StraTT> under the `agda/` directory.

First, assume a type of `Levels` along with a well-founded order `_<_` on them. The proof of well-foundedness `wf` has

<sup>2</sup>[coq/incr.v:DTyping\\_incr](#)

type  $\forall (k : \text{Level}) \rightarrow \text{Acc } k$ , where  $\text{Acc } k$  is the usual accessibility predicate and  $\text{acc} <$  its constructor.<sup>3</sup>

Now, the most direct way to model universes is as follows.<sup>4</sup>

```

data U (k : Level) : Set
el :  $\forall k \rightarrow U k \rightarrow \text{Set}$ 
data U k where
   $\hat{U} : U k$ 
   $\hat{I} : U k$ 
   $\hat{\Pi} : \forall j \rightarrow j < k \rightarrow (A : U j) \rightarrow$ 
     $(B : \text{el } j A \rightarrow U k) \rightarrow U k$ 
el k  $\hat{U} = U k$ 
el k  $\hat{I} = \perp$ 
el k  $(\hat{\Pi} j j < k A B) = (x : \text{el } j A) \rightarrow \text{el } k (B x)$ 
    
```

The universe  $U k$  contains the code for itself  $\hat{U}$ , the code for the empty type  $\hat{I}$ , and the code for dependent functions  $\hat{\Pi}$ , containing a strictly smaller level  $j$ , the code for its domain at the smaller level, and a function that produces a code for its codomain at the same level given an element of the interpretation of the code of the domain. The interpretation  $\text{el } k$  interprets each code as expected, though in contrast to usual inductive–recursive models, the interpretation of  $\hat{U}$  in  $U k$  isn't some smaller  $U j$ , but rather  $U k$  itself.

Agda will reject this inductive–recursive definition for not being strictly positive, because in the type of the  $B$  argument of  $\hat{\Pi}$ , Agda thinks  $U$  could appear in a negative position as the result of  $\text{el}$ . However, we know that only a strictly smaller  $U j$  will be returned by virtue of well-foundedness of the levels, so this definition is morally valid. To convince Agda of this, we adapt the technique from Kovács [16] and parameterize  $U$  by  $U <$  and  $\text{el} <$ . These parameters represent universes at strictly smaller levels and interpretation functions that can only be used on these strictly smaller universes.

```

data U' k (U< :  $\forall \{j\} \rightarrow j < k \rightarrow \text{Set}$ )
  (el< :  $\forall \{j\} (j < k : j < k) \rightarrow U' k j < k \rightarrow \text{Set}$ )
  : Set
el' :  $\forall k (U< : \forall \{j\} \rightarrow j < k \rightarrow \text{Set})$ 
  (el< :  $\forall \{j\} (j < k : j < k) \rightarrow U' k j < k \rightarrow \text{Set}$ )
   $\rightarrow U' k U< \text{el} < \rightarrow \text{Set}$ 
    
```

With this change, the  $A$  argument of  $\hat{\Pi}$  has type  $U < j < k$ , while the  $B$  argument has type  $\text{el} < j < k A \rightarrow U' k U < j < k$ , no longer violating strict positivity. We tie the knot by defining the top-level  $U <$  and  $\text{el} <$  by induction over accessibility predicates on levels, then finally instantiate the predicates by well-foundedness in  $U$  and  $\text{el}$ .

```

U< :  $\forall \{k\} \rightarrow \text{Acc } k \rightarrow \forall \{j\} \rightarrow j < k \rightarrow \text{Set}$ 
el< :  $\forall \{k\} (p : \text{Acc } k) \{j\} (j < k : j < k)$ 
   $\rightarrow U < p j < k \rightarrow \text{Set}$ 
U< (acc< f)  $\{j\} j < k =$ 
   $U' j (U < (f j < k)) (\text{el} < (f j < k))$ 
el< (acc< f)  $\{j\} j < k =$ 
    
```

<sup>3</sup>agda/Acc.agda

<sup>4</sup>agda/direct-model.agda

```

el' j (U < (f j < k)) (el < (f j < k))
U :  $\forall k \rightarrow \text{Set}$ 
U k = U' k (U < (wf k)) (el < (wf k))
el :  $\forall k \rightarrow U k \rightarrow \text{Set}$ 
el k = el' k (U < (wf k)) (el < (wf k))
    
```

To correctly model the cumulativity of subStratT, we also need to show that universes and codes are cumulative as well. More precisely, we prove that given a code in  $U j$ , it can be lifted to a larger universe  $U k$ , and given an element in the interpretation of the smaller code, we can produce an element in the interpretation of the lifted code.

```

lift :  $\forall \{j k\} \rightarrow j < k \rightarrow U j \rightarrow U k$ 
el $\rightarrow$  :  $\forall \{j k\} \rightarrow (j < k : j < k) \rightarrow$ 
   $\forall u \rightarrow \text{el } j u \rightarrow \text{el } k (\text{lift } j < k u)$ 
    
```

The proofs of these cumulativity lemmas are slightly involved due to having to deal with accessibility proofs, which further requires assuming that  $\text{Acc } k$  is a mere proposition. These proofs and the full definitions of  $U'$  and  $\text{el}'$  can also be found in the Agda files.<sup>5</sup>

### 3 StratT and floating functions

We have found that subStratT alone is insufficiently expressive, with some examples being unexpectedly untypeable and others being simply clunky to work with. The full StratT system therefore extends the subsystem with a separate non-dependent function type, written  $A \rightarrow B$ , that does not have the same level restriction on the domain as the dependent function type.

$$\begin{array}{c}
 \text{DT-ARROW} \\
 \frac{\Delta; \Gamma \vdash A :^k \star \quad \Delta; \Gamma \vdash B :^k \star}{\Delta; \Gamma \vdash A \rightarrow B :^k \star} \\
 \text{DT-ABSTM} \\
 \frac{\Delta; \Gamma \vdash A :^k \star \quad \Delta; \Gamma \vdash B :^k \star}{\Delta; \Gamma, x :^k A \vdash b :^k B} \\
 \frac{\Delta; \Gamma \vdash \lambda x. b :^k A \rightarrow B}{\Delta; \Gamma \vdash \lambda x. b :^k A \rightarrow B} \\
 \text{DT-APPTM} \\
 \frac{\Delta; \Gamma \vdash b :^k A \rightarrow B \quad \Delta; \Gamma \vdash a :^k A}{\Delta; \Gamma \vdash b a :^k B}
 \end{array}$$

Figure 2. Typing rules (nondependent 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 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

<sup>5</sup>agda/model.agda

to be a higher level while the latter doesn't. From the perspective of Stratified System F, STRTT merely generalizes stratified type polymorphism over types to include term polymorphism.

We say that the domain of these nondependent function types *floats* because unlike dependent function types, it isn't fixed to some particular level. The interaction between nondependent functions and cumulativity is where this becomes interesting. Given a function  $f$  of type  $A \rightarrow B$  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  $k$ , 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. This behaviour is why the consistency model from Section 2.1 can't straightforwardly be extended to accommodate nondependent function types. We examine the issue in detail in Section 6.4.

### 3.1 Examples

**The identity function.** Here's 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 constant name.)

$$\begin{aligned} \text{id}_0 &: {}^1 \Pi X : {}^0 \star. \Pi x : {}^0 X. X \\ \text{id}_0 X x &:= x \end{aligned}$$

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.

Although the level of the domain of  $\text{id}_0$  is fixed at 0, we can displace the constant by 1.

$$\begin{aligned} \text{id}_1 &: {}^2 \Pi X : {}^1 \star. \Pi x : {}^1 X. X \\ \text{id}_1 &:= \text{id}_0^1 \end{aligned}$$

Since we have cumulativity, we would expect to be able to apply  $\text{id}_1$  to itself. This is possible with a typical cumulative universe hierarchy, such as in Coq. In the below definition, since  $(\text{forall } (X : \text{Type}@\{u0\}), X \rightarrow X)$  can be assigned type  $\text{Type}@\{u1\}$ , it can be used as the first argument to  $\text{id}_1$ . The second argument must then have type  $(\text{forall } (X : \text{Type}@\{u0\}), X \rightarrow X)$ . While  $\text{id}_1$  itself doesn't have this type, we can  $\eta$ -expand it to a function that does, since  $\text{Type}@\{u0\}$  is a subtype of  $\text{Type}@\{u1\}$  and thus  $X$  of the former type can be passed into a function that takes the latter.

Universe u0 u1.

Constraint u0 <= u1.

Definition id1 (X : Type@{u1}) (x : X) : X := x.

Definition idid1 : forall X, X -> X :=  
id1 (forall (X : Type@{u0}), X -> X)  
(fun X x => id1 X x).

However, the analogous definition applying  $\text{id}_1$  to itself doesn't type check! The problematic subterm is bolded in red below.

$$\begin{aligned} \text{idid}_1 &: {}^2 \Pi X : {}^0 \star. \Pi x : {}^0 X. X \\ \text{idid}_1 &:= \text{id}_1 (\Pi X : {}^0 \star. \Pi x : {}^0 X. X) (\lambda X. \lambda x. \text{id}_1 X x) \end{aligned}$$

The type  $\Pi X : {}^0 \star. \Pi x : {}^0 X. X$  is well typed at level 1, but the term  $\lambda X. \lambda x. \text{id}_1 X x$  is only well typed with that type (again via subsumption) at level 2, so the latter can't be applied as the second argument to  $\text{id}_1$ , which is fixed at level 1.

Here is where floating nondependent function type comes to use. Since the second argument isn't depended upon in the type, we can assign the identity function as follows.

$$\begin{aligned} \text{id} &: {}^1 \Pi X : {}^0 \star. X \rightarrow X \\ \text{id } X x &:= x \end{aligned}$$

Now the argument  $x$  and the function body are both at level 1 without requiring subsumption. The argument and body of a nondependent function having the same level is key to typing the self-application.

$$\begin{aligned} \text{idid} &: {}^2 \Pi X : {}^0 \star. X \rightarrow X \\ \text{idid} &:= \text{id}^1 (\Pi X : {}^0 \star. X \rightarrow X) (\lambda X. \lambda x. \text{id}^1 X x) \end{aligned}$$

Displacing  $\text{id}$  by 1, we can then pass in the type  $\Pi X : {}^0 \star. X \rightarrow X$ , which has level 1, followed by  $\lambda X. \lambda x. \text{id}^1 X x$ , which has level 2, yielding a final term at level 2.

**Decidable types.** Floating nondependent function types are similarly crucial for type constructors. Later in Section 5 we'll consider datatypes with parameters, but for now, consider the following Church encoding [2] of decidable types, which additionally uses negation defined as implication into the empty type.

$$\begin{aligned} \text{neg} &: {}^0 \star \rightarrow \star \\ \text{neg } X &:= X \rightarrow \perp \\ \text{Dec} &: {}^1 \star \rightarrow \star \\ \text{Dec } X &:= \Pi Z : {}^0 \star. (X \rightarrow Z) \rightarrow (\text{neg } X \rightarrow Z) \rightarrow Z \\ \text{yes} &: {}^1 \Pi X : {}^0 \star. X \rightarrow \text{Dec } X \\ \text{yes } X x &:= \lambda Z. \lambda f. \lambda g. f x \\ \text{no} &: {}^1 \Pi X : {}^0 \star. \text{neg } X \rightarrow \text{Dec } X \\ \text{no } X nx &:= \lambda Z. \lambda f. \lambda g. g nx \end{aligned}$$

The  $\text{yes } X$  constructor decides  $X$  by a witness, while the  $\text{no } X$  constructor decides  $X$  by its refutation. We're able to

show that deciding a given type is irrefutable.<sup>6</sup>

$$\begin{aligned} \text{irrDec} &: \Pi X :^0 \star. \text{neg} (\text{Dec } X) \\ \text{irrDec } X \text{ } ndec &:= ndec (\text{no } X (\lambda x. ndec (\text{yes } X x))) \end{aligned}$$

Without the nondependent function type, `neg` and `Dec` would be forced produce types at higher levels. The corresponding constructors `yes'` and `no'`, omitted below, would also have higher levels.

$$\begin{aligned} \text{neg}' &: ^1 \Pi X :^0 \star. \star \\ \text{neg}' X &:= \Pi x :^0 X. \perp \\ \text{Dec}' &: ^3 \Pi X :^0 \star. \star \\ \text{Dec}' X &:= \Pi Z :^0 \star. \Pi yz :^2 (\Pi x :^0 X. Z). \\ &\quad \Pi nz :^2 (\Pi nx :^1 \text{neg}' X. Z). Z \end{aligned}$$

Every dependent quantification in the domain increases the overall level, so the smallest level that can be assigned to `Dec' X` is 3, since it takes a function eliminating a `no'`, which is a function taking the negation of `X`, which itself is a function from `X`. We can continue on to write the corresponding type of `irrDec'`, displacing `neg'` as needed, but the body will no longer type check against it.

$$\begin{aligned} \text{irrDec}' &: ^5 \Pi X :^0 \star. \text{neg}'^4 (\text{neg}'^3 (\text{Dec}' X)) \\ \text{irrDec}' X \text{ } ndec &:= ndec (\text{no}' X (\lambda x. ndec (\text{yes}' X x))) \end{aligned}$$

The level of the function `ndec` of type `neg'^3 (Dec' X)` is now 4, which is too high to be used in the argument of `no'`; if we displace `yes'` and `no'`, then the level of the argument of `ndec` will in turn be too high to fit.

**Leibniz equality.** Although nondependent functions can often benefit from a floating domain, sometimes we don't want the domain to float. In some examples, 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. Here, we turn to a simple application of dependent types with Leibniz equality [17, 21] to demonstrate such a situation.

$$\begin{aligned} \text{eq} &: ^1 \Pi X :^0 \star. X \rightarrow X \rightarrow \star \\ \text{eq } X x y &:= \Pi P :^0 X \rightarrow \star. P x \rightarrow P y \end{aligned}$$

An equality `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.

$$\begin{aligned} \text{refl} &: ^1 \Pi X :^0 \star. \Pi x :^0 X. \text{eq } X x x \\ \text{refl } X x P px &:= px \end{aligned}$$

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.

$$\begin{aligned} \text{isProp} &: ^0 \star \rightarrow \star \\ \text{isProp } X &:= \Pi x^0 X. \Pi y^0 X. \text{eq } X x y \end{aligned}$$

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; informally, stratification propagates dependency information not only from the codomain, but also from the function body.

$$\begin{aligned} \text{isProp} &: ^1 \Pi X :^0 \star. \star \\ \text{isProp } X &:= \Pi x :^0 X. \Pi y :^0 X. \text{eq } X x y \end{aligned}$$

Going one further, we can define a predicate `isSet` stating that `X` is an h-set [32], or that its equalities are mere propositions, by using a displaced `isProp`, which also raises the overall level. Once again, despite the type of `isSet` not being an actual dependent function type, here we need to fix the level of the domain.

$$\begin{aligned} \text{isSet} &: ^2 \Pi X :^0 \star. \star \\ \text{isSet } X &:= \Pi x :^0 X. \Pi y :^0 X. \text{isProp}^1 (\text{eq } X x y) \end{aligned}$$

## 4 Syntactic metatheory

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. The proof scripts are available at <https://github.com/plclub/StraTT> under the `coq/` directory.

**Strengthening.** The key idea of this type system design is that stratification levels delineate judgements. 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 states that variables from higher levels can always be removed from the context and that contexts can be truncated at any level.

Formally, we define the *restriction* operation, written  $[\Gamma]^k$ , that filters out all assumptions from the context with level greater than  $k$ .

**Lemma 4.1** (Restriction). <sup>7</sup> If  $\Delta \vdash \Gamma$  then  $\Delta \vdash [\Gamma]^k$  for any  $k$ , and if  $\Delta; \Gamma \vdash a :^k A$  then  $\Delta; [\Gamma]^k \vdash a :^k A$ .

**Weakening and Narrowing.** We can extend the ordering between levels,  $j \leq k$ , to an ordering between contexts,  $\Gamma_1 \leq \Gamma_2$ . At the same time, we also incorporate the idea of weakening into this relation. Stronger contexts have higher levels and fewer assumptions.

<sup>6</sup>Note this differs from irrefutability of the law of excluded middle, `neg (neg (ΠX :^0 ⋆. Dec X))`, which cannot be proven constructively.

<sup>7</sup>`coq/ctx.v:DSig_DCtx_DTyping_restriction`

$$\begin{array}{c}
\boxed{\Gamma_1 \leq \Gamma_2} \\
\text{S-NIL} \\
\hline
\emptyset \leq \emptyset
\end{array}
\quad
\begin{array}{c}
\text{S-CONS} \\
j \leq k \\
\Gamma_1 \leq \Gamma_2 \\
\hline
\Gamma_1, x :^j A \leq \Gamma_2, x :^k A
\end{array}
\quad
\begin{array}{c}
\text{(Context ordering)} \\
\text{S-WEAK} \\
\Gamma_1 \leq \Gamma_2 \\
\hline
\Gamma_1, x :^k A \leq \Gamma_2
\end{array}$$

This ordering is contravariant in the typing judgement: we can lower the context without destroying typeability. This result subsumes a standard weakening lemma.

**Lemma 4.2** (Weakening/Narrowing).<sup>8</sup> If  $\Delta; \Gamma \vdash a :^k A$  and  $\Delta \vdash \Gamma'$  and  $\Gamma' \leq \Gamma$  then  $\Delta; \Gamma' \vdash a :^k A$ .

Note that a restricted context could be stronger than the original because it could contain fewer variable assumptions.

**Lemma 4.3** (Restriction subsumption).<sup>9</sup>  $\Gamma \leq [\Gamma]^k$ .

**Substitution and regularity.** The substitution operation must respect levels.

**Lemma 4.4** (Substitution).<sup>10</sup> If  $\Delta; \Gamma_1, x :^j B, \Gamma_2 \vdash a :^k A$  and  $\Delta; \Gamma_1 \vdash b :^j B$  then  $\Delta; \Gamma_1, \Gamma_2 \{b/x\} \vdash a\{b/x\} :^k A\{b/x\}$ .

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

Typing judgements themselves ensure the well-formedness of their subcomponents. Note that if a term type checks, its type can be typed at the same level.

**Lemma 4.5** (Regularity).<sup>11</sup> If  $\Delta; \Gamma \vdash a :^k A$  then  $\vdash \Delta$  and  $\Delta \vdash \Gamma$  and  $\Delta; \Gamma \vdash A :^k \star$

Because our type system includes the non-syntax-directed rule **T-CONV**, the proof of this lemma depends on several inversion lemmas (not shown).

**Displacement.** Derivations can be displaced wholesale. For example, if we have a derivation

$$f :^1 \Pi x :^0 A. B, x :^0 A \vdash f x :^1 B$$

we can add 1 to each level in the judgement to get a new derivation.

$$f :^2 \Pi x :^1 A. B, x :^1 A \vdash f x :^2 B$$

More generally, we can add any level displacement  $i$  to each level in the judgement.

$$f :^{1+i} \Pi x :^i A. B, x :^i A \vdash f x :^{1+i} B$$

This is a generalization of displaceability in an empty context to include displacement of contexts, written as  $\Gamma^{+i}$ , where  $(\Gamma, x :^k A)^{+i} = \Gamma^{+i}, x :^{k+i} A^{+i}$ .

**Lemma 4.6** (Displacement).<sup>12</sup> If  $\Delta; \Gamma \vdash a :^k A$  then  $\Delta; \Gamma^{+j} \vdash a^{+j} :^{j+k} A^{+j}$ .

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

**Floating.** Sometimes we would like to use a dependent function, like  $f$  above, at a higher level, without displacing its type. The cumulativity rule 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 does not 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 a special case, 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, suppose we have the following derivation.

$$f :^j \Pi x :^i A. B, x :^i A \vdash f x :^j B$$

This structure would also make sense at a higher level  $k$ , where we have also raised the level of  $f$ .

$$f :^k \Pi x :^i A. B, x :^i A \vdash f x :^k B$$

However, we can only raise the level of variables, like  $f$ , that are at the *same* level as the entire judgement. For example, we cannot raise  $x$  which is at some lower level  $i$  because then it would be invalid as an argument to  $f$ .

To prove this formally, we must work with judgements that do not have any assumptions above the current level. We know from strengthening that these assumptions aren't needed by the derivation, but their types could mention the variables that are being raised, and we want to ensure that the resulting context is well formed. Therefore, we use the restriction operation to first discard the irrelevant variables.

Furthermore, to raise the level of certain levels, we introduce the *floating* operation on contexts  $\uparrow_j^k \Gamma$  that raises assumptions in  $\Gamma$  at level  $j$  to some higher level  $k$  without displacing their types.

**Lemma 4.7** (Restricted Floating).<sup>13</sup> If  $\Delta; \Gamma \vdash a :^j A$  and  $j \leq k$  then  $\Delta; \uparrow_j^k([\Gamma]^j) \vdash a :^k A$ .

The proof of the lemma above requires the following two observations. First, that floating starting from a lower number is stronger than floating from a higher number.

**Lemma 4.8.**<sup>14</sup> Say  $i \leq j$  and  $j \leq k$ , then  $\uparrow_j^k([\Gamma]^j) \leq \uparrow_i^k([\Gamma]^i)$

Second, if we have a strict ordering between  $i$  and  $j$ , then it doesn't matter what the higher number floats to because they won't be present in the lower restriction.

<sup>8</sup>coq/ctx.v:DTyping\_SubG

<sup>9</sup>coq/restrict.v:SubG\_restrict

<sup>10</sup>coq/subst.v:DTyping\_subst

<sup>11</sup>coq/ctx.v:DCtx\_DSig, coq/inversion.v:DTyping\_DCtx,

coq/ctx.v:DTyping\_regularity

<sup>12</sup>coq/ctx.v:DTyping\_incr

<sup>13</sup>coq/restrict.v:DTyping\_float\_restrict

<sup>14</sup>coq/restrict.v:SubG\_float\_leq

**Lemma 4.9.**<sup>15</sup> Say  $i < j$  and  $i \leq k_2$ , then for any  $k_1$ ,  $\uparrow_j^{k_1}(\lceil \Gamma \rceil^j) \leq \uparrow_i^{k_2}(\lceil \Gamma \rceil^i)$

**Type Safety.** We have shown that this language satisfies the preservation (i.e. subject reduction) and progress lemmas, where  $\boxed{\Delta \vdash a \rightsquigarrow b}$  consists of  $\beta$ - and  $\delta$ -reduction with call-by-name evaluation.

**Lemma 4.10 (Preservation).**<sup>16</sup> If  $\Delta; \Gamma \vdash a :^k A$  and  $\Delta \vdash a \rightsquigarrow a'$  then  $\Delta; \Gamma \vdash a' :^k A$ .

The case for  $\beta$ -reduction of nondependent functions requires the expressiveness of the restricted floating lemma above. In this case, we have a derivation of the form  $\Gamma \vdash (\lambda x. a) b :^k B$ , and we would like to show that the term after reduction  $\Gamma \vdash a\{b/x\} :^k B$  is well typed.

Our inversion lemmas give us the typing of the function and argument at some level  $k$ .

$$\Gamma \vdash \lambda x. a :^k A \rightarrow B \quad \Gamma \vdash b :^k A$$

Inverting the first derivation again, we find that because of cumulativity, for some  $j \leq k$  we have  $\Gamma, x :^j A_1 \vdash a :^j A_2$ .

We cannot directly use the substitution lemma to show the result because  $x$  needs to be at level  $j$ , but the argument may be at some potentially higher level  $k$ . Therefore, to complete the proof, we use the restricted floating lemma above to modify the derivation above so that we have the one required for substitution.

$$\uparrow_j^k(\lceil \Gamma \rceil^j), x :^k A_1 \vdash a :^k A_2$$

We can then weaken this derivation to the original context

$$\Gamma, x :^k A_1 \vdash a :^k A_2$$

and then, with the help of rule **T-CONV** to convert the types  $A_2$  to  $B$  and  $A$  to  $A_1$ , conclude the desired result via our substitution lemma.

Because we have not mechanized the metatheory of our equivalence relation  $A \equiv B$ , the preservation proof relies on standard properties about this relation [1] that we state as axioms in the development. Levels do not play a role in definitional equality.

**Proposition 4.1 (Injectivity for nondependent function types).**<sup>17</sup> If  $A_1 \rightarrow A_2 \equiv A \rightarrow B$  then  $A_1 \equiv A$  and  $A_2 \equiv B$ .

**Proposition 4.2 (Injectivity for dependent function types).**<sup>18</sup> If  $\Pi x :^j_1 A_1. B_1 \equiv \Pi x :^j_2 A_2. B_2$  then  $A_1 \equiv A_2$ ,  $j_1 = j_2$ , and  $B_1 \equiv B_2$ .

**Proposition 4.3 (Consistency of definitional equality).**<sup>19</sup> If  $A \equiv B$  then  $A$  and  $B$  do not have different head forms.

Preservation for the reflexive, transitive closure of reduction also follows from preservation for reduction.

**Corollary 4.1 (Preservation).**<sup>20</sup> If  $\Delta; \Gamma \vdash a :^k A$  and  $\Delta \vdash a \rightsquigarrow^* a'$  then  $\Delta; \Gamma \vdash a' :^k A$ .

The progress lemma follows in the usual way.

**Lemma 4.11 (Progress).**<sup>21</sup> If  $\Delta; \emptyset \vdash a :^k A$  then either  $a$  is a value or there is some  $b$  such that  $\Delta \vdash a \rightsquigarrow b$ .

Aside from the assumptions about definitional equality stated above, the case of `absurd(a)` in this proof also requires that the language be logically consistent, which we assume as an axiom in our Coq development.<sup>22</sup>

## 5 Prototype implementation with level inference and datatypes

To explore the expressiveness of StraTT, we have implemented a prototype type checker, which can be found at <https://github.com/plclub/StraTT> under the `impl/` directory. This implementation is based on `pi-forall` [33], a simple bidirectional type checker for a dependently-typed programming language.

**Annotation inference.** 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 linear 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.

As an example, recall the `idid` example from Section 3.1.

$$\text{idid} :^2 \Pi X :^0 \star. X \rightarrow X$$

$$\text{idid} := \text{id}^1 (\Pi X :^0 \star. X \rightarrow X) (\lambda X. \lambda x. \text{id}^1 X x)$$

All annotations can be left out, and the exact same levels and displacements will be inferred.

$$\text{idid}' : \Pi X : \star. X \rightarrow X$$

$$\text{idid}' := \text{id} (\Pi X : \star. X \rightarrow X) (\lambda X. \lambda x. \text{id} X x)$$

For clarity, in the remaining examples in this section, we always write annotations explicitly. However, each of these examples also type check with their annotations removed.

More generally, we don't yet know whether a most general set of displacements and level annotations always exists and is discoverable by an SMT solver. Although we've found that our simple approach to inference doesn't always produce the best solution in some constructed examples, in future work we seek to address the inference question with more sophisticated tools.

<sup>15</sup>`coq/restrict.v:SubG_float_lt`

<sup>16</sup>`coq/typesafety.v:Reduce_Preservation`

<sup>17</sup>`coq/axioms.v:DEquiv_Arrow_inj1, DEquiv_Arrow_inj2`

<sup>18</sup>`coq/axioms.v:DEquiv_Pi_inj1, DEquiv_Pi_inj2`

<sup>19</sup>`coq/axioms.v:ineq_*`

<sup>20</sup>`coq/typesafety.v:WHNF_Preservation`

<sup>21</sup>`coq/typesafety.v:progress`

<sup>22</sup>`coq/axioms.v:empty_Bottom`

**Datatypes.** 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 to ensure consistency is possible but orthogonal to stratification and thus out of scope for this work. In this section and the next, the examples we provide will always satisfy strict positivity and structural termination.

Revisiting an example from Section 3.1, we can define Dec as a datatype.

```
data Dec (X : ★) :0 ★ where
  Yes :0 X → Dec X
  No  :0 neg X → Dec X
```

The lack of annotation on the parameter indicates that it's a floating domain, so that  $\lambda X. \text{Dec } X$  can be assigned type  $\star \rightarrow \star$  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.

```
decDNE :1  $\Pi X :0 \star. \text{Dec } X \rightarrow \text{neg } (\text{neg } X) \rightarrow X$ 
decDNE X dec nn := case dec of
  Yes y  $\Rightarrow$  y
  No x   $\Rightarrow$  absurd(nn x)
```

By the level annotation on the function, we know that *dec* and *nnx* 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.

**Datatype displacement.** Datatypes and their constructors, like constants, can be displaced as well, uniformly raising the levels of their types. Consider now a *Box* type which stores in it a term of a fixed lower level.

```
data Box (X :0 ★) :1 ★ where
  MkBox :1  $\Pi x :0 X. \text{Box } X$ 
```

This time, the presence of the level 0 annotation on the parameter indicates that it's fixed, which also allows the constructor argument *x* to be fixed at level 0. Displacing a *Box* by 1 then raises the fixed parameter level to 1, and its sole constructor would be a *MkBox* displaced by 1 which destructs to yield a term at level 1. The following map over a box displaced by 1 with a function at level 1 demonstrates

this behaviour.

```
map :2  $\Pi X :1 \star. \Pi Y :1 \star. \Pi f :1 X \rightarrow Y. \text{Box}^1 X \rightarrow \text{Box}^1 Y$ 
map X Y f box := case box of
  MkBox1 x  $\Rightarrow$  MkBox1 (f x)
```

### 5.1 Extended example: dependent pairs

Along with *Dec* and *Box*, we include with our implementation a variety of common datatypes and associated functions, such as dependent pairs, logical connectives, booleans, optionals, naturals, lists, finite sets, and vectors.<sup>23</sup> Here, we take a closer look at dependent pairs to further explore the behaviour and limitations of StraTT.

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.

```
data NPair (X :0 ★) (P : X → ★) :1 ★ where
  MkPair :1  $\Pi x :0 X. P x \rightarrow \text{NPair } X P$ 
nfst :2  $\Pi X :0 \star. \Pi P :1 X \rightarrow \star. \text{NPair } X P \rightarrow X$ 
nfst X P p := case p of MkPair x y  $\Rightarrow$  x
nsnd :2  $\Pi X :0 \star. \Pi P :1 X \rightarrow \star. \Pi p :1 \text{NPair } X P. P (\text{nfst } X P p)$ 
nsnd X P p := case p of MkPair x y  $\Rightarrow$  y
```

Due to stratification, the projections unfortunately need to be defined at level 2 to accommodate dependently quantifying over the *P* 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.

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 define a universe of propositions *NPair*  $\star$  **isProp**, as you'll recall that the predicate has type  $\Pi X :<sup>0</sup> \star. \star$  at level 1.

```
data DPair (X :0 ★) (P :  $\Pi x :0 X. \star$ ) :1 ★ where
  MkPair :1  $\Pi x :0 X. P x \rightarrow \text{DPair } X P$ 
dfst :2  $\Pi X :0 \star. \Pi P :1 (\Pi x :0 X. \star). \text{DPair } X P \rightarrow X$ 
dfst X P p := case p of MkPair x y  $\Rightarrow$  x
dsnd :2  $\Pi X :0 \star. \Pi P :1 (\Pi x :0 X. \star). \Pi p :1 \text{DPair } X P.$ 
      case p of MkPair x y  $\Rightarrow$  P x
dsnd X P p := case p of 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

<sup>23</sup>[impl/pi/README.pi](https://github.com/imp1/pi/README.pi)

by matching on the pair. Now we’re able to define  $\text{DPair} \star$   $\text{isProp}$ .

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  $\text{isSet}$ , which has type  $\Pi X :^0 \star. \star$  at level 2: the universe of sets  $\text{DPair} \star \text{isSet}$  is also well typed at level 2.

Unfortunately, the first projection  $\text{dfst}$  can no longer be used on an element of this pair, since the predicate is now at level 2, nor can its displacement  $\text{dfst}^1$ , 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  $\text{isProp}$ ), one with levels 0 and 2 (as in the case of  $\text{isSet}$ ), and so on.

## 6 On consistency

In this section, we delve into the design of  $\text{StraTT}$  and the implementation as they relate to logical consistency, *i.e.* the absence of a closed inhabitant of  $\perp$ .

### 6.1 Level annotations

The lack of level annotations on unstratified nondependent function types lends to them their flexibility with respect to cumulativity. A declared function  $f :^0 A \rightarrow B$  taking and returning a term at level 0 can, by subsumption, be used as a function taking and returning a term at *any* higher level so long as the input and output levels match. It may be tempting to remove the level annotation on dependent function types as well, so that they enjoy the same flexibility as long as the output level is strictly greater than the input level, but this recovers impredicativity, thus defeating the purpose of stratification. Supposing the level annotations are removed and that we have some well-typed function type  $\Pi x : \star. B$  at level 1, the following derivation is valid.

$$\frac{f :^1 \Pi x : \star. B \vdash f :^2 \Pi x : \star. B \quad f :^1 \Pi x : \star. B \vdash \Pi x : \star. B :^1 \star}{f :^1 \Pi x : \star. B \vdash f (\Pi x : \star. B) :^2 B\{\Pi x : \star. B/x\}}$$

Without the level annotation, the application rule for dependent functions now applies merely whenever the argument is typed at a level strictly lower than the function, allowing our function type to be substituted into its own codomain. With both impredicativity and type-in-type, this

system would be no different from an unstratified system with type-in-type, allowing us to derive an inconsistency.

### 6.2 Constructor levels

In the implementation, a datatype definition is valid if, among the other rules discussed, the level of the constructors cannot be higher than that of the datatype itself. If the level were allowed to be higher, while regularity wouldn’t be violated, it would yet again be possible to derive an inconsistency. We demonstrate this with a variant of Burali-Forti’s paradox [4] concerning the simultaneous well-foundedness and non-well-foundedness of particular datatype  $U$ <sup>24,25</sup>

**data**  $U :^0 \star$  **where**

$\text{MkU} :^1 \Pi X :^0 \star. (X \rightarrow U) \rightarrow U$

While  $\text{MkU}$  is assigned level 1, we consider the possibility of assigning level 0 to its type  $U$ . Note that this definition is strictly positive, so we aren’t using any tricks relying on negative datatypes. Next, we define a well-foundedness predicate for  $U$ .

**data**  $\text{WF} :^2 \Pi u :^1 U. \star$  **where**

$\text{MkWF} :^2 \Pi X :^0 \star. \Pi f :^1 X \rightarrow U. (\Pi x :^1 X. \text{WF} (f x))$   
 $\rightarrow \text{WF} (\text{MkU} X f)$

This definition is strictly positive as well, while the constructor’s level is *not* greater than that of the datatype. It’s easy to show that all  $U$  are well founded.

$\text{wf} :^2 \Pi u :^1 U. \text{WF} u$

$\text{wf} u := \text{case } u \text{ of}$

$\text{MkU } X f \Rightarrow \text{MkWF } X f (\lambda x. \text{wf} (f x))$

However, with  $U$  typeable at level 0, we’re able to construct an inhabitant that is provably *not* well founded.

$\text{loop} :^1 U$

$\text{loop} := \text{MkU } U (\lambda u. u)$

$\text{nwfLoop} :^2 \text{WF loop} \rightarrow \perp$

$\text{nwfLoop } \text{nwfLoop} := \text{case } \text{nwfLoop} \text{ of}$

$\text{MkWF } X f h \Rightarrow \text{nwfLoop} (h \text{ loop})$

In the branch of  $\text{nwfLoop}$ , by pattern matching on the type of the scrutinee,  $X$  is bound to  $U$  and  $f$  to  $\lambda u. u$ , so  $h \text{ loop}$  correctly has type  $\text{WF loop}$ . Note that this definition would also pass the usual structural termination check, since the recursive call is done on a subargument from  $h$ . Then  $\text{nwfLoop} (\text{wf loop})$  is an inhabitant of the empty type. By restricting the level of the constructor to the level of its datatype, the level of  $U$  must be 1, so the key problematic definition  $\text{loop}$  is immediately disallowed.

A similar phenomenon occurs when trying to implement a variant of Russell’s paradox [28], making use of the same  $U$

<sup>24</sup>This example was provided by Stephen Dolan in private correspondence.

<sup>25</sup>[impl/pi/WFU.pi](#)

above (with its level as 1).<sup>26,27</sup> 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.

$$\begin{aligned} \text{regular} & :^1 U \rightarrow \star \\ \text{regular } u & = \text{case } u \text{ of} \\ \text{MkU } X f & \Rightarrow \Pi x :^0 X. (f \ x = \text{MkU } X f) \rightarrow \perp \end{aligned}$$

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.

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

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  $\text{MkU}^2$ .

### 6.3 Hurkens' paradox

Although we've seen that stratification thwarts the paradoxes in Section 6.2, that paradox leverages the properties of datatypes and recursive functions, which we haven't formalized. Here, we'll turn to the failure of Hurkens' paradox 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.

```
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.
```

<sup>26</sup>This formulation is due to Paolo Capriotti [9], and the Agda implementation can be found at <https://github.com/agda/agda/blob/master/test/Succeed/Russell.agda>.

<sup>27</sup>[impl/pi/Russell.pi](#)

then the definitions check up to  $M$ . The corresponding StraTT code, too, checks up to  $M$ , as verified in the implementation.<sup>28</sup> Displacement is sufficient to handle situations in which polymorphism was needed, and explicitly indicates where.

$$\begin{aligned} P & :^0 \star \rightarrow \star \\ P X & := X \rightarrow \star \\ U & :^1 \star \\ U & := \Pi X :^0 \star. (P (P X) \rightarrow X) \rightarrow P (P X) \\ \text{tau} & :^1 P (P U) \rightarrow U \\ \text{tau } t X f p & := t (\lambda s. p (f (s X f))) \\ \text{sig} & :^2 U^1 \rightarrow P (P U) \\ \text{sig } s & := s U \text{ tau} \\ \text{Delta} & :^2 P U^1 \\ \text{Delta } y & := (\Pi p :^1 P U. \text{sig } y p \rightarrow p (\text{tau } (\text{sig } y))) \rightarrow \perp \\ \text{Omega} & :^3 U \\ \text{Omega} & := \text{tau } (\lambda p. \Pi x :^2 U^1. \text{sig } x p \rightarrow p (\lambda X. x X)) \\ M & :^4 \Pi x :^3 U^2. \text{sig}^1 x \text{Delta} \rightarrow \text{Delta}^1 x \\ M x s d & := d \text{Delta } s (\lambda p. d (\lambda y. p (\text{tau } (\text{sig } y)))) \\ D & :^3 \star \\ D & := \Pi p :^1 P U. (\Pi x :^1 U. \text{sig } x p \rightarrow p x) \rightarrow p \text{Omega} \end{aligned}$$

The next definition  $D$  doesn't type check, since  $\text{sig}$  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.

### 6.4 Proof attempt

The main difficulty in proving consistency via a semantic model of StraTT is the interpretation of the nondependent function type in the presence of cumulativity. Continuing with the framework from Section 2.1, recall that for each level  $k$  we have a collection  $U_k$  of codes of types of StraTT and a mapping  $\text{el}_k$  from codes into the semantic domain. Now let  $\eta$  be a mapping from variables into the semantic domain, and consider the following fragment of an interpretation from StraTT terms into the semantic domain.

$$\begin{aligned} \llbracket x \rrbracket^k \eta & = \eta(x) \\ \llbracket \Pi x :^j A. B \rrbracket^k \eta & = \hat{\Pi}_k(j < k, \llbracket A \rrbracket^j \eta, x \mapsto \llbracket B \rrbracket^k \langle \eta, x \rangle) \\ \llbracket A \rightarrow B \rrbracket^k \eta & = \hat{\rightarrow}_k(\llbracket A \rrbracket^k \eta, \llbracket B \rrbracket^k \eta) \\ \llbracket \perp \rrbracket^k \eta & = \hat{\perp}_k \end{aligned}$$

<sup>28</sup>[impl/pi/Hurkens.pi](#) (no annotations), [impl/pi/HurkensAnnot.pi](#) (all annotations)

Here, we have codes  $\hat{\Pi}$ ,  $\hat{\rightarrow}_k$ , and  $\hat{\downarrow}_k$  in  $U_k$ . Our fundamental lemma then states the following, disregarding any issues regarding weakening of contexts and  $\eta$  mappings.

**Lemma 6.1** (Fundamental lemma). Let  $\eta$  be a mapping such that for every  $x : ^j A \in \Gamma$ ,  $\eta(x) \in \text{el}_j(\llbracket A \rrbracket^j \eta)$ . If  $\Gamma \vdash a : ^k A$ , then  $\llbracket a \rrbracket^k \eta \in \text{el}_k(\llbracket A \rrbracket^k \eta)$ .

Consistency arises from the fundamental lemma combined with the appropriate definition of  $\text{el}_k(\hat{\downarrow}_k)$ , which in the Agda model would be Agda’s empty type. The fundamental lemma is proven by induction on the typing derivation. In the rule **DT-VAR** case, we have that  $\llbracket x \rrbracket^k \eta = \eta(x) \in \text{el}_j(\llbracket A \rrbracket^j)$ , while we need to show that  $\llbracket x \rrbracket^k \eta \in \text{el}_k(\llbracket A \rrbracket^k)$ . We thus need a lemma stating that cumulativity is preserved.

**Lemma 6.2** (Preservation of cumulativity). Suppose  $j \leq k$ . If  $a \in \text{el}_j(\llbracket A \rrbracket^j \eta)$  then  $a \in \text{el}_k(\llbracket A \rrbracket^k \eta)$ .

We proceed by induction on the structure of  $A$ . With some unfolding, we can see the case of  $\Pi x : ^j A. B$  poses no issue, since the level of its domain is fixed. In the case of  $A \rightarrow B$ , we need to show that if  $f \in \text{el}_j(\hat{\rightarrow}_j(\llbracket A \rrbracket^j, \llbracket B \rrbracket^j))$ , then  $f \in \text{el}_k(\hat{\rightarrow}_k(\llbracket A \rrbracket^k, \llbracket B \rrbracket^k))$ , with the induction hypotheses stating that if  $a \in \text{el}_j(\llbracket A \rrbracket^j)$  then  $a \in \text{el}_k(\llbracket A \rrbracket^k)$ , and similarly for  $B$ . Since terms of type  $A \rightarrow B$  behave like functions, we expect that the interpretation of  $\hat{\rightarrow}$  codes behaves like a function space. However, to prove our goal, we require that they not be contravariant in the domain with respect to cumulativity, as expected from a function space, but *covariant!*

Concretely, in the Agda model, supposing we have an interpretation function  $\llbracket \_ \rrbracket k \eta : U k$  given a representation of a StraTT term  $A$ , a level  $k$ , and a mapping  $\eta$ , we would need to show that given a function

$$f : \text{el } j (\llbracket A \rrbracket j \eta) \rightarrow \text{el } j (\llbracket B \rrbracket j \eta)$$

and an element  $a : \text{el } k (\llbracket A \rrbracket k \eta)$ , we have some element in  $\text{el } k (\llbracket B \rrbracket k \eta)$ , but the application  $f a$  is ill typed, and we can’t demote  $a$  to a lower level.

One solution would be to change the interpretation of nondependent functions to bake in cumulativity, something along the lines of  $A \rightarrow B = \hat{\rightarrow}_k (i \geq k, \llbracket A \rrbracket^i, \llbracket B \rrbracket^i)$ , but this would prevent  $U_k$  and  $\text{el}_k$  from being well defined by well-founded induction on  $k$ .

We are thus at an impasse among stratified dependent functions, floating nondependent functions, and cumulativity, all of which are fundamental features of StraTT.

## 7 Related Work

StraTT is directly inspired from Leivant’s stratified polymorphism [7, 18, 19], which developed from the ramified polymorphic typed  $\lambda$ -calculus briefly introduced by Statman [29]. Stratified System F, a slight modification of the original system, has since been used to demonstrate a normalization proof technique using hereditary substitution [11], which

in turn has been mechanized in Coq as a case study for the Equations package [20]. More recently, an interpreter of an intrinsically-typed Stratified System F has been mechanized in Agda [31], where stratification levels are interpreted as Agda’s universe levels. Similarly, Hubers and Morris [14] model in Agda a stratification of  $R_\omega$ , which is a System  $F_\omega$  with row types. Meanwhile, our system of level displacement comes from McBride’s crude but effective stratification [23, 24], following Hou (Favonia) et al. [13] and specializing the displacement algebra to the naturals.

## 8 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 a challenge for future work, we have provided extensive evidence supporting our conjecture of consistency 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, but demonstrating that inference is complete remains future work. We also leave formally ensuring that our rules for datatypes don’t violate existing metatheoretical properties as future work.

We have chosen level displacement over level polymorphism for its simplicity, expressivity, and extensibility. Although we fix levels to the naturals in both the proof mechanization (to take advantage of Coq’s arithmetic tactics) and the Haskell implementation (to enable solving level metavariable constraints), it should be possible to use any displacement algebra in the sense of Hou (Favonia) et al. [13]. This opens future exploration with, for instance, transfinite levels, as are used in transfinite stratified polymorphism [7]. Even so, nothing precludes adopting explicit level polymorphism in place of displacement. This raises interesting questions, such as at what level a term like  $\Lambda \ell. \Pi x : ^\ell A. B$  should be typed, or whether a consistency proof would be further complicated by needing to model such terms.

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] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. North-Holland, 1984.
- [2] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed  $\lambda$ -programs on term algebras. *Theoretical Computer Science*, 39: 135–154, 1985.
- [3] Edwin C. Brady. IDris — Systems Programming Meets Full Dependent Types. In *Programming Languages Meets Program Verification*, pages 43–54. Association for Computing Machinery, January 2011. ISBN 9781450304870. doi:10.1145/1929529.1929536.
- [4] Cesare Burali-Forti. Una questione sui numeri transfiniti. *Rendiconti del Circolo matematico di Palermo*, 11, 1897.
- [5] 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>.
- [6] The Coq Development Team. The Coq Proof Assistant, January 2022. URL <https://coq.github.io/doc/v8.15/refman>.
- [7] Normal Danner and Daniel Leivant. Stratified polymorphism and primitive recursion. *Mathematical Structures in Computer Science*, 9(4): 507–522, 1999. doi:10.1017/S0960129599002868.
- [8] 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. ISBN 978-3-319-21400-9, 978-3-319-21401-6. doi:10.1007/978-3-319-21401-6\_26.
- [9] Dominique Devriese. [Agda] Simple contradiction from type-in-type, March 2013. URL <https://lists.chalmers.se/pipermail/agda/2013/005164.html>.
- [10] Peter Dybjer. Internal type theory. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs*, volume 1158, pages 120–134. Springer Berlin Heidelberg, 1996. ISBN 978-3-540-61780-8 978-3-540-70722-6. doi:10.1007/3-540-61780-9\_66.
- [11] 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>.
- [12] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l’arithmétique d’ordre supérieur*. PhD dissertation, Université Paris VII, 1972.
- [13] 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.
- [14] 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. ISSN 2475-1421. doi:10.1145/3607843.
- [15] 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. ISBN 978-3-540-49178-1.
- [16] 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. ISBN 978-3-95977-218-1. doi:10.4230/LIPIcs.CSL.2022.28. URL <https://drops.dagstuhl.de/opus/volltexte/2022/15748>.
- [17] Gottfried Wilhelm Leibniz. Discours de métaphysique, 1686.
- [18] 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.
- [19] Daniel Leivant. Finitely stratified polymorphism. *Information and Computation*, 93(1):93–113, 1991. ISSN 0890-5401. doi:10.1016/0890-5401(91)90053-5. Selections from 1989 IEEE Symposium on Logic in Computer Science.
- [20] 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. doi:10.4204/EPTCS.185.5. URL <https://hal.inria.fr/hal-01248807>.
- [21] Per Martin-Löf. A theory of types, 1971.
- [22] Per Martin-Löf. An intuitionistic theory of types, 1972.
- [23] Conor McBride. Crude but Effective Stratification, 2002. URL <https://personal.cis.strath.ac.uk/conor.mcbride/Crude.pdf>.
- [24] Conor McBride. Crude but Effective Stratification, 2011. URL <https://mazzo.li/epilogue/index.html%3Fp=857&cpage=1.html>.
- [25] 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>.
- [26] John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, pages 408–425, Berlin, Heidelberg, 1974. Springer Berlin Heidelberg. ISBN 978-3-540-37819-8.
- [27] 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. ISBN 978-3-540-38891-3.
- [28] Bertrand Russell. *The Principles of Mathematics*. Cambridge University Press, 1903.
- [29] 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.
- [30] Nikhil Swamy, Cătălin Hrișcu, 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. ISBN 978-1-4503-3549-2. doi:10.1145/2837614.2837655.
- [31] 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/>.
- [32] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. URL <https://homotopytypetheory.org/book>.
- [33] Stephanie Weirich. Implementing Dependent Types in pi-forall, 2023. URL <https://arxiv.org/abs/2207.02129>.

## A Well-formedness and equality

$\boxed{\vdash \Delta}$	(Signature formation)	
D-EMPTY	D-CONS	
$\frac{}{\vdash \emptyset}$	$\frac{\begin{array}{l} \vdash \Delta \\ \Delta; \emptyset \vdash A :^k \star \\ \Delta; \emptyset \vdash a :^k A \\ x \notin \text{dom } \Delta \end{array}}{\vdash \Delta, x :^k A := a}$	
$\boxed{\Delta \vdash \Gamma}$	(Context formation)	
DG-EMPTY	DG-CONS	
$\frac{\vdash \Delta}{\Delta \vdash \emptyset}$	$\frac{\begin{array}{l} \Delta \vdash \Gamma \\ \Delta; \Gamma \vdash A :^k \star \\ x \notin \text{dom } \Gamma \\ x \notin \text{dom } \Delta \end{array}}{\Delta \vdash \Gamma, x :^k A}$	
$\boxed{\Delta \vdash a \equiv b}$	(Definitional equality)	
DE-REFL	DE-SYM	DE-TRANS
$\frac{}{\Delta \vdash a \equiv a}$	$\frac{\Delta \vdash b \equiv a}{\Delta \vdash a \equiv b}$	$\frac{\Delta \vdash a \equiv b \quad \Delta \vdash b \equiv c}{\Delta \vdash a \equiv c}$
DE-BETA	DE-ETA	
$\frac{}{\Delta \vdash (\lambda x. b) a \equiv b\{a/x\}}$	$\frac{b' = b x}{\Delta \vdash \lambda x. b' \equiv b}$	
DE-DELTA	DE-ARROW	
$\frac{x :^k A := a \in \Delta}{\Delta \vdash x^i \equiv a^{+i}}$	$\frac{\Delta \vdash A \equiv A' \quad \Delta \vdash B \equiv B'}{\Delta \vdash A \rightarrow B \equiv A' \rightarrow B'}$	
DE-PI	DE-ABS	
$\frac{\Delta \vdash A \equiv A' \quad \Delta \vdash B \equiv B'}{\Delta \vdash \Pi x :^k A. B \equiv \Pi x :^k A'. B'}$	$\frac{}{\Delta \vdash \lambda x. b \equiv \lambda x. b'}$	
DE-APP	DE-ABSURD	
$\frac{\Delta \vdash a \equiv a' \quad \Delta \vdash b \equiv b'}{\Delta \vdash b a \equiv b' a'}$	$\frac{}{\Delta \vdash \text{absurd}(b) \equiv \text{absurd}(b')}$	

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