Bounded First-Class Universe Levels in Dependent Type Theory

Jonathan Chan
University of Pennsylvania, Philadelphia, USA

Stephanie Weirich
University of Pennsylvania, Philadelphia, USA

Abstract

In dependent type theory, being able to refer to a type universe as a term itself increases its expressive power, but requires mechanisms in place to prevent Girard's paradox from introducing logical inconsistency in the presence of type-in-type. The simplest mechanism is a hierarchy of universes indexed by a sequence of levels, typically the naturals. To improve reusability of definitions, they can be made level polymorphic, abstracting over level variables and adding a notion of level expressions. For even more expressive power, level expressions can be made first-class as terms themselves, and level polymorphism is subsumed by dependent functions quantifying over levels. Furthermore, bounded level polymorphism provides more expressivity by being able to explicitly state constraints on level variables. While semantics for first-class levels with constraints are known, syntax and typing rules have not been explicitly written down. Yet pinning down a well-behaved syntax is not trivial; there exist prior type theories with bounded level polymorphism that fail to satisfy subject reduction. In this work, we design an explicit syntax for a type theory with bounded first-class levels, parametrized over arbitrary well-founded sets of levels. We prove the metatheoretic properties of subject reduction, type safety, consistency, and canonicity, entirely mechanized from syntax to semantics in Lean.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases type theory, universes, universe polymorphism

Supplementary Material *Software (source code)*: https://github.com/ionathanch/TTBFL
archived at swh:1:dir:8f18b01234056282a037b3d835e97df2b5050b29

1 Introduction

Dependent type theories are common foundations for proof assistants, where theorems are manipulated as types and their proofs as terms. Types are often treated as terms themselves, providing a uniform mechanism for working with both; for example, quantifying over predicates is no different from quantifying over functions, as predicates are functions that return types. To merge types and terms, we need a type of types, or a *universe*, which itself must be a term with a type.

Girard [10] showed that a type-in-type axiom makes dependent type theory logically inconsistent: if the type of a universe is itself, then all types are inhabited, rendering the type theory useless as a tool for proving. Therefore, Martin-Löf stratified the universe in his type theory (MLTT) [16] into a countably infinite hierarchy of universes $\mathsf{U}_0 : \mathsf{U}_1 : \mathsf{U}_2 : \ldots$ indexed by *universe levels* spanning the naturals. Many contemporary proof assistants based on dependent types feature such a hierarchy, such as Rocq [4], Agda [17], Lean [7], and F* [20].

Having only a concrete universe hierarchy, however, limits the reusability of definitions that are not inherently tied to particular universe levels. For example, the identity function $\mathsf{id} : \Pi A : \mathsf{U}_i.\ A \to A$ would need to be redefined for each universe level i at which it is needed. Universe level polymorphism addresses this issue by abstracting over level variables, used to index universes alongside concrete levels. Its simplest form is prenex level polymorphism, introduced by Harper and Pollack [11], which restricts the abstraction to top-level definitions.

Courant [5] extends their implicit system to an explicit system with (in)equality constraints, level operators, and level expressions. This extension is implemented in Rocq [19].

If we disallow recursive definitions that vary in the level, uses of prenex-polymorphic definitions can be specialized to level-monomorphic terms. Favonia, Angiuli, and Mullanix note that it "is as consistent as standard (monomorphic) type theory [...] because any given proof can only mention finitely many universes", and show consistency using this idea [12].

If level quantification is added as a type former directly to the type theory, we obtain higher-rank level polymorphism, where level-polymorphic terms can be passed as arguments to functions. For instance, Bezem, Coquand, Dybjer, and Escardó introduce such a type theory (referred to here as BCDE) with level constraints [3]. Going further, rather than keeping universe levels distinct from terms, we can make them first class by defining level expressions as a subset of terms, and add a type of levels; such levels are found in Agda. Level quantification is subsumed by dependent functions whose domain is this level type. The codomain can also be the level type, which describes functions that compute levels.

First-class universe levels are known to be logically consistent. In particular, Kovács [14] gives a semantic model for a type theory TTFL, which features first-class levels and an ordering relation $<$ on them. The model is given as categories with families (cwfs) [8], mostly mechanized in Agda using induction–recursion, and supports features such as level constraints, maxima of levels, and induction on levels.

The syntax of TTFL is considered to be the initial model in the category of cwfs, but an explicit syntax and typing rules are not given, and proving initiality even for MLTT is a colossal task [6]. Furthermore, while a syntax may satisfy semantic properties such as logical consistency, it may not necessarily satisfy desirable syntactic properties. In particular, BCDE's semantics can conceivably be viewed as that of TTFL without making levels first class, yet its syntax fails to satisfy subject reduction.

In this work, we give an end-to-end account of first-class levels in type theory, beginning with an explicit syntax and typing rules, and proving that they satisfy desirable metatheoretic properties. Our contributions are as follows:

- We present **TTBFL**, a dependent type theory with bounded, first-class universe levels. Our bounds differ from level constraints in that they are inherent to the type of a level, rather than a separate predicate on them, which prevents failure of subject reduction. Examples in the next section build up from monomorphic levels to level polymorphism before we proceed to the formal definition of the type theory in Section 3.
- We prove subject reduction (*i.e.* preservation) in Section 4, an improvement upon the metatheoretic properties of BCDE. We also prove progress and thus type safety, which is important if we also want to use the language for writing programs that evaluate. An example is implementing proof assistants in themselves, as is (partially) done in Lean and undergoing work for Rocq [18].
- Using a syntactic logical relation, we prove logical consistency and canonicity via the fundamental soundness theorem in Section 5. Consistency ensures that the type theory is suitable as a basis for logical reasoning in a proof assistant, while canonicity ensures that closed terms evaluate to the values we expect. Normalization of open terms remains an open problem (Section 6).

All results are mechanized in Lean. The development consists of under 1700 lines of code, which can be found in the supplementary materials at `https://github.com/ionathanch/TTBFL`. The definitions and theorems in this paper are hyperlinked to the corresponding Lean files.

As our system is intentionally very minimal, we discuss some further extensions in Section 7, including level operators and subtyping. We conclude with future work in Section 8.

2 Motivation

To motivate the range of features in TTBFL, we look at examples starting from monomorphic universe levels and build up to first-class levels and bounding in this section. Although not found in our minimal language, these examples use dependent pairs, propositional equality, the naturals, and lists for more illuminating examples.

Let us start by revisiting the identity function and its type, supposing $\mathsf{U}_0 : \mathsf{U}_1 : \ldots \mathsf{U}_\omega$, with a limit universe ω at the top, which will come in handy later.

$$\mathsf{Id} : \mathsf{U}_1 \qquad\qquad \mathsf{id} : \mathsf{Id}$$
$$\mathsf{Id} \coloneqq \Pi A : \mathsf{U}_0.\ A \to A \qquad\qquad \mathsf{id} \coloneqq \lambda A : \mathsf{U}_0.\ \lambda x : A.\ x$$

This identity function is polymorphic over types in U_0, but not over universes, so the self application $\mathsf{id}\ \mathsf{Id}\ \mathsf{id}$ is ill typed. More generally, if we want to reuse a definition at different universe levels, it would need to be redefined for every level needed. If we introduce prenex polymorphism of universe levels, where top-level definitions are permitted to be polymorphic, we can write a universe polymorphic identity function that can be instantiated at different levels and self-applied.

$$\mathsf{Id} : \forall i.\ \mathsf{U}_{i+1} \qquad\qquad \mathsf{id} : \forall i.\ \mathsf{Id}\ [i]$$
$$\mathsf{Id} \coloneqq \Lambda i.\ \Pi A : \mathsf{U}_i.\ A \to A \qquad\qquad \mathsf{id} \coloneqq \Lambda i.\ \lambda A : \mathsf{U}_i.\ \lambda x : A.\ x$$

Now, the expression $\mathsf{id}\ [1]\ (\mathsf{Id}\ [0])\ (\mathsf{id}\ [0])$ is well typed. A definition can also be polymorphic over multiple levels, such as the constant function that takes two arguments but always returns the first. For this, we need a binary least upper bound operator $\sqcup$ on levels.

$$\mathsf{Const} : \forall i.\ \forall j.\ \mathsf{U}_{(i \sqcup j)+1} \qquad\qquad \mathsf{const} : \forall i.\ \forall j.\ \mathsf{Const}\ [i]\ [j]$$
$$\mathsf{Const} \coloneqq \Lambda i.\ \Lambda j.\ \Pi A : \mathsf{U}_i.\ \Pi B : \mathsf{U}_j.\ A \to B \to A \qquad\qquad \mathsf{const} \coloneqq \Lambda i.\ \Lambda j.\ \lambda A.\ \lambda B.\ \lambda x.\ \lambda y.\ x$$

The universe in which $\mathsf{Const}\ [i]\ [j]$ lives is $(i \sqcup j) + 1$, because its universe must contain the universes U_i and U_j over which it quantifies. As more level variables get involved, the algebraic expressions on levels becomes increasingly complex. But the precise universe in which Const lives is not as important as knowing that it lives in *some* greater universe, which is all that is needed to prevent type-in-type inconsistencies. This can be expressed by bounded level quantification, simplifying level expressions at the cost of an additional level variable. We use the limit level ω to allow k to range over all other levels.

$$\mathsf{Const} : \forall k < \omega.\ \forall i < k.\ \forall j < k.\ \mathsf{U}_k$$
$$\mathsf{Const} \coloneqq \Lambda k.\ \Lambda i.\ \Lambda j.\ \Pi A : \mathsf{U}_i.\ \Pi B : \mathsf{U}_j.\ A \to B \to A$$

While nonrecursive prenex level polymorphism can be monomorphized away, this is not the case once we introduce recursive definitions whose recursive calls may vary in the level. This lets us define universes with levels incremented by fixed amount, *i.e.* U_{k+n}.

$$\mathsf{incr} : \forall k < \omega.\ \mathsf{Nat} \to \mathsf{U}_\omega$$
$$\mathsf{incr}\ k\ \mathsf{zero} \coloneqq \mathsf{U}_k$$
$$\mathsf{incr}\ k\ (\mathsf{succ}\ n) \coloneqq \mathsf{incr}\ n\ [k+1]$$

Generalizing from prenex level polymorphism to higher-rank level polymorphism affords even more reusability. One application is when axioms are explicitly assumed as local hypotheses instead of globally axiomatized to restrict their usage to only where they are really needed. An example is function extensionality, whose type is level polymorphic.

$$\mathsf{FunExt} : \forall k < \omega.\ \forall i < k.\ \forall j < k.\ \mathsf{U}_k$$
$$\mathsf{FunExt} \coloneqq \Lambda k.\ \Lambda i.\ \Lambda j.\ \Pi A : \mathsf{U}_i.\ \Pi B : (A \to \mathsf{U}_j).$$
$$\Pi f : (\Pi x : A.\ B\ x).\ \Pi g : (\Pi x : A.\ B\ x).\ (\Pi x : A.\ f\ x = g\ x) \to f = g$$

Suppose we wished to prove that function extensionality for functions with two arguments

at different universe levels follows from assuming FunExt. Using only prenex polymorphism, we would need two separate instantiations, once for its application to the functions of type $\Pi x : A.\, B\ x \to C\ x$, and once for its application to the functions of type $B\ x \to C\ x$.

$$\begin{aligned}&\mathsf{lemma} : \forall l < \omega.\, \forall i < l.\, \forall j < l.\, \forall k < l.\, (\mathsf{FunExt}\ [l]\ [i]\ [j \sqcup k]) \to (\mathsf{Funext}\ [l]\ [j]\ [k]) \to \\ &\qquad \Pi A : \mathsf{U}_i.\, \Pi B : (A \to \mathsf{U}_j).\, \Pi C : (A \to \mathsf{U}_k). \\ &\qquad \Pi f : (\Pi x : A.\, B\ x \to C\ x).\, \Pi g : (\Pi x : A.\, B\ x \to C\ x). \\ &\qquad (\Pi x : A.\, \Pi y : B\ x.\, f\ x\ y = g\ x\ y) \to f = g \\ &\mathsf{lemma} \coloneqq \Lambda l.\, \Lambda i.\, \Lambda j.\, \Lambda k.\, \lambda fe1.\, \lambda fe2. \ldots\end{aligned}$$

Once more universe levels get involved, instantiating up front every possible use becomes unwieldy. With higher-rank polymorphism, we can quantify over a polymorphic function extensionality principle once and for all, and instantiate its levels within the proof as needed.

$$\begin{aligned}&\mathsf{lemma} : (\forall k < \omega.\, \forall i < k.\, \forall j < k.\, \mathsf{FunExt}\ [i]\ [j]\ [k]) \to \\ &\qquad \forall l < \omega.\, \forall i < l.\, \forall j < l.\, \forall k < l.\, \Pi A : \mathsf{U}_i.\, \Pi B : (A \to \mathsf{U}_j).\, \Pi C : (A \to \mathsf{U}_k). \\ &\qquad \Pi f : (\Pi x : A.\, B\ x \to C\ x).\, \Pi g : (\Pi x : A.\, B\ x \to C\ x). \\ &\qquad (\Pi x : A.\, \Pi y : B\ x.\, f\ x\ y = g\ x\ y) \to f = g \\ &\mathsf{lemma} \coloneqq \lambda fe.\, \Lambda l.\, \Lambda i.\, \Lambda j.\, \Lambda k. \ldots.\end{aligned}$$

With higher-rank level polymorphism, a level-polymorphic type itself must live in some universe, which is often that of the bounding level. Coming back to the identity function, we can impose a bound on its level by bounded quantification, and use the bound for the universe. Self-applications such as $\mathsf{id}\ [2]\ [1]\ (\mathsf{Id}\ [1])\ (\mathsf{id}\ [1])$ still hold.

$$\begin{aligned}&\mathsf{Id} : \forall j < \omega.\, \mathsf{U}_j && \mathsf{id} : \forall j < \omega.\, \mathsf{Id}\ [j] \\ &\mathsf{Id} \coloneqq \Lambda j.\, \forall i < j.\, \Pi A : \mathsf{U}_i.\, A \to A && \mathsf{id} \coloneqq \Lambda j.\, \Lambda i.\, \lambda A : \mathsf{U}_i.\, \lambda x : A.\, x\end{aligned}$$

So far, our notions of level polymorphism treat levels as syntactically separate from terms, with special level operators $\cdot + 1$ and $\cdot \sqcup \cdot$. Consequently, if we want more general ways to compute level expressions, we must add them as primitives to the language. If we instead make levels first class, we are then able to manipulate and store them as terms. Bounded level quantifications are subsumed by ordinary dependent types whose domain is the type of all levels bounded by some strictly greater level $\mathsf{Level{<}}\ k$. An example application is computing the least upper bound level from a list of levels and types of that level.

$$\begin{aligned}&\mathsf{lub} : \mathsf{List}\ (\Sigma i : \mathsf{Level{<}}\ \omega.\, \mathsf{U}\ i) \to \mathsf{Level{<}}\ \omega \\ &\mathsf{lub}\ \mathsf{nil} \coloneqq 0 \\ &\mathsf{lub}\ (\mathsf{cons}\ (i, A)\ As) \coloneqq i \sqcup (\mathsf{lub}\ As)\end{aligned}$$

This level computation can be used to turn a list of types and their levels into an n-ary tuple with a precise level. This is a technique used, for instance, by Escot and Cockx in generic programming to represent level-polymorphic inductive types [9].

$$\begin{aligned}&\mathsf{Interp} : \Pi As : \mathsf{List}\ (\Sigma i : \mathsf{Level{<}}\ \omega.\, \mathsf{U}\ i).\, \mathsf{U}\ (\mathsf{lub}\ As) \\ &\mathsf{Interp}\ \mathsf{nil} \coloneqq \top \\ &\mathsf{Interp}\ (\mathsf{cons}\ (i, A)\ As) \coloneqq A \times (\mathsf{Interp}\ As)\end{aligned}$$

Various proof assistants with universe level polymorphism implement different subsets of these features. Lean and F* have prenex polymorphism with successor and least upper bound operators. Rocq has prenex polymorphism along with level (in)equality declarations, but no other operators. Agda has first-class levels and the two level operator, but no level constraints. In TTBFL, we include bounded first-class levels, but omit the two level operators for simplicity, opting to treat them as straightforward potential extensions.

$$
\begin{aligned}
i, j &::= \texttt{<concrete universe levels>} \\
x, y, z &::= \texttt{<term variables>} \\
a, b, c, A, B, C, k, \ell &::= x \mid i \mid \Pi x : A.\, B \mid \lambda x : A.\, b \mid b\ a \mid \bot \mid \mathsf{absurd}_A\ b \mid \mathsf{U}\ k \mid \mathsf{Level{<}}\ \ell \\
\Gamma, \Delta &::= \cdot \mid \Gamma, x : A
\end{aligned}
$$

Figure 1 Syntax ⟨syntactics.lean:Term,Ctxt⟩

$$
\textsc{Nil}\ \frac{}{\vdash \cdot} \qquad
\textsc{Cons}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A : \mathsf{U}\ k}{\vdash \Gamma, x : A} \qquad
\textsc{Var}\ \frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash x : A} \qquad
\textsc{Pi}\ \frac{\Gamma \vdash A : \mathsf{U}\ k \qquad \Gamma, x : A \vdash B : \mathsf{U}\ k}{\Gamma \vdash \Pi x : A.\, B : \mathsf{U}\ k}
$$

$$
\textsc{Lam}\ \frac{\Gamma \vdash A : \mathsf{U}\ k \qquad \Gamma \vdash \Pi x : A.\, B : \mathsf{U}\ k \qquad \Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x : A.\, b : \Pi x : A.\, B} \qquad
\textsc{App}\ \frac{\Gamma \vdash b : \Pi x : A.\, B \qquad \Gamma \vdash a : A}{\Gamma \vdash b\ a : B[x \mapsto a]}
$$

$$
\textsc{Mty}\ \frac{\Gamma \vdash \mathsf{U}\ k : \mathsf{U}\ \ell}{\Gamma \vdash \bot : \mathsf{U}\ k} \qquad
\textsc{Abs}\ \frac{\Gamma \vdash A : \mathsf{U}\ k \qquad \Gamma \vdash b : \bot}{\Gamma \vdash \mathsf{absurd}_A\ b : A} \qquad
\textsc{Conv}\ \frac{\Gamma \vdash a : A \qquad \Gamma \vdash B : \mathsf{U}\ k \qquad A \equiv B}{\Gamma \vdash a : B}
$$

$$
\textsc{E-Beta}\ \frac{}{(\lambda x : A.\, b)\ a \equiv b[x \mapsto a]} \qquad
\textsc{E-Refl}\ \frac{}{a \equiv a} \qquad
\textsc{E-Sym}\ \frac{a \equiv b}{b \equiv a} \qquad
\textsc{E-Trans}\ \frac{a \equiv b \qquad b \equiv c}{a \equiv c} \qquad \ldots
$$

Figure 2 Typing and selected equality rules (no universes or levels) ⟨typing.lean:Wtf,Eqv⟩

3 A minimal type theory with bounded first-class universe levels

TTBFL is a Church-style type theory *à la* Russell, where terms may have type annotations, and there is no separate typing judgement for well-formedness of types. To keep the type theory minimal, it contains only dependent functions, an empty type, predicative universes, and bounded universe levels. By convention, we use a, b, c for terms, A, B, C for types, and k, ℓ for level terms. The syntax is presented in Figure 1; we additionally use $A \to B$ as sugar for nondependent functions $\Pi x : A.\, B$ where x does not occur in B. While the mechanization uses de Bruijn indexing and simultaneous substitutions, this paper presents the syntax in nominal form for clarity, and we omit the details of manipulating substitutions for concision. We write single substitutions of a variable x in a term b by another term a as $b[x \mapsto a]$.

The type theory is parametrized over a cofinal woset of levels, *i.e.* a set of levels that are well founded, totally ordered, and each have some strictly larger level; these properties are required when modelling the type theory. Instances of such sets include the naturals $0, 1, 2, \ldots$, as well as the naturals extended by one limit ordinal ω and its successors $\omega+1, \omega+2, \ldots$. We continue to use these concrete levels for our examples. These metalevel levels are internalized directly in system as terms i.

We begin first with the basic rules that don't concern universes or levels in Figure 2, consisting of a context well-formedness judgement $\boxed{\vdash \Gamma}$, a typing judgement $\boxed{\Gamma \vdash a : A}$, and an untyped definitional equality $\boxed{a \equiv b}$. We use β-conversion as our equality, and omit the usual congruence rules. Unusually, rule Lam includes well-typedness premises of both the function's type and the domain type alone. The former is necessary to strengthen the induction hypotheses when proving the fundamental soundness theorem, and the latter to strengthen them when proving subject reduction. We later prove admissible a rule Lam'

$$\frac{\Gamma \vdash k : \mathsf{Level}\!<\ell}{\Gamma \vdash \mathsf{U}\,k : \mathsf{U}\,\ell}\ \textsc{Univ} \qquad \frac{\Gamma \vdash \mathsf{U}\,k_1 : \mathsf{U}\,\ell_1 \qquad \Gamma \vdash k_0 : \mathsf{Level}\!<\ell_0}{\Gamma \vdash \mathsf{Level}\!<k_0 : \mathsf{U}\,k_1}\ \textsc{Level}< \qquad \frac{\vdash \Gamma \qquad i < j}{\Gamma \vdash i : \mathsf{Level}\!<j}\ \textsc{Lvl}$$

$$\frac{\Gamma \vdash k_1 : \mathsf{Level}\!<k_2 \qquad \Gamma \vdash k_2 : \mathsf{Level}\!<k_3}{\Gamma \vdash k_1 : \mathsf{Level}\!<k_3}\ \textsc{Trans} \qquad \frac{\Gamma \vdash A : \mathsf{U}\,k \qquad \Gamma \vdash k : \mathsf{Level}\!<\ell}{\Gamma \vdash A : \mathsf{U}\,\ell}\ \textsc{Cumul}$$

Figure 3 Typing rules (universes and levels)

that omits the first premise. The other typing rules are otherwise typical.

The rules relating to universes and levels are given in Figure 3. By rule Lvl, we can view the type constructor Level< as a restricted internalization of the order on levels. Quantifications and abstractions over a level variable must be bounded by some level expression, which cannot be the variable itself since it is not in the scope of its own type. In contrast, if we had more general level constraint types, it would be possible to declare a looping constraint $x < x$. The level type itself can be typed at any universe by rule Level< regardless of its bounding level. For example, we can construct a derivation for $\cdot \vdash \mathsf{Level}\!<\!2 : \mathsf{U}0$ solely knowing that $\cdot \vdash 2 : \mathsf{Level}\!<3$, $\cdot \vdash \mathsf{U}\,0 : \mathsf{U}\,1$, which follow from $0 < 1$ and $2 < 3$.

Rule Trans internalizes transitivity of the order on levels, which is now required since levels are terms in general and not only concrete levels. For example, we can construct a derivation for $x : \mathsf{Level}\!<\omega, y : \mathsf{Level}\!<x \vdash x : \mathsf{Level}\!<\omega$, where the levels x, y are variables. Rule Cumul is a cumulativity rule that permits lifting a type from one universe to a higher universe. This rule is weaker than a full subtyping rule that accounts for contravariance in the domain and covariance in the codomain of function types. Therefore, for instance, $f : \mathsf{U}\,2 \to \mathsf{U}\,0 \vdash f : \mathsf{U}\,1 \to \mathsf{U}\,1$ does *not* hold. Nonetheless, cumulativity allows us to instead type the η-expansion $f : \mathsf{U}\,2 \to \mathsf{U}\,0 \vdash \lambda x : \mathsf{U}\,1.\, f\,x : \mathsf{U}\,1 \to \mathsf{U}\,1$.

Finally, rule Univ asserts that a universe at level k lives in the universe at level ℓ when k is strictly bounded by ℓ. Allowing universes with general level terms and not just concrete levels to be well typed is what permits typing level-polymorphic types. For example, the level-polymorphic identity function type $\Pi x : \mathsf{Level}\!<\omega.\, \Pi y : \mathsf{U}\,x.\, y \to y$ is typeable. $\mathsf{Level}\!<\omega$ can be assigned an arbitrary type by rule Level, $\mathsf{U}\,x$ has type $\mathsf{U}\,\omega$ by rule Univ and rule Var, and y can be assigned type $\mathsf{U}\,\omega$ transitively via rules Trans and Var. Then the entire term has type $\mathsf{U}\,\omega$ by repeated application of rule Pi.

4 Type safety

Type safety is proven using standard syntactic methods to show progress and preservation (*i.e.* subject reduction). In essence, closed, well-typed terms evaluate (if they terminate) to values, which are type formers and constructors, defined below. The proof is standard, so we omit most details, listing only some of the key lemmas required.

$v ::= i \mid \Pi x : A.\, B \mid \lambda x : A.\, b \mid \bot \mid \mathsf{U}\,k \mid \mathsf{Level}\!<\ell$ ⟨safety.lean:Value⟩

4.1 Reduction and conversion

Rather than working directly with β-reduction, we use parallel reduction $\boxed{a \Rightarrow b}$, defined in Figure 4, and its reflexive, transitive closure $\boxed{a \Rightarrow^* b}$, into which call-by-name evaluation embeds. Similarly, instead of definitional equality, we use conversion $\boxed{a \Leftrightarrow b}$, which is defined in terms of parallel reduction. We begin with simple lemmas about parallel reduction.

$$\text{P-Beta}\ \frac{b \Rightarrow b' \qquad a \Rightarrow a'}{(\lambda x : A.\ b)\ a \Rightarrow b'[x \mapsto a']} \qquad \text{P-Pi}\ \frac{A \Rightarrow A' \qquad B \Rightarrow B'}{\Pi x : A.\ B \Rightarrow \Pi x : A'.\ B'} \qquad \text{P-Lam}\ \frac{A \Rightarrow A' \qquad b \Rightarrow b'}{\lambda x : A.\ b \Rightarrow \lambda x : A'.\ b'} \qquad \text{P-Univ}\ \frac{k \Rightarrow k'}{\mathsf{U}\ k \Rightarrow \mathsf{U}\ k'}$$

$$\text{P-App}\ \frac{b \Rightarrow b' \qquad a \Rightarrow a'}{b\ a \Rightarrow b'\ a'} \qquad \text{P-Abs}\ \frac{A \Rightarrow A' \qquad b \Rightarrow b'}{\mathsf{absurd}_A\ b \Rightarrow \mathsf{absurd}_{A'}\ b'} \qquad \text{P-Level<}\ \frac{\ell \Rightarrow \ell'}{\mathsf{Level{<}}\ \ell \Rightarrow \mathsf{Level{<}}\ \ell'} \qquad \text{P-Var}\ \frac{}{x \Rightarrow x} \qquad \text{P-Lvl}\ \frac{}{i \Rightarrow i} \qquad \text{P-Mty}\ \frac{}{\bot \Rightarrow \bot}$$

Figure 4 Parallel reduction rules ⟨reduction.lean:Par,Pars⟩

▶ **Definition 1** (Conversion). ⟨reduction.lean:Conv⟩
$a \Leftrightarrow b$ iff there exists a c such that $a \Rightarrow^* c$ and $b \Rightarrow^* c$

▶ **Lemma 2** (Substitution (p.r.)). ⟨reduction.lean:parsSubst⟩
If $a \Rightarrow^* a'$ and $b \Rightarrow^* b'$, then $b[x \mapsto a] \Rightarrow^* b'[x \mapsto a']$.

▶ **Lemma 3** (Construction (p.r.)). ⟨reduction.lean:pars{β,Pi,Abs,$\mathcal{U}$,App,Exf,Lvl}⟩
Analogous constructors of parallel reduction hold for its reflexive, transitive closure, e.g. if $b \Rightarrow^* b'$ and $a \Rightarrow^* a'$, then $(\lambda x : A.\ b)\ a \Rightarrow^* b'[x \mapsto a']$.

▶ **Lemma 4** (Inversion (p.r.)). ⟨reduction.lean:pars{Pi,Abs,$\mathcal{U}$,App,Exf,Lvl,Lof,Mty}Inv⟩
If $v \Rightarrow^* c$, then c is also a value of the same syntactic shape such that the reduction is congruent, e.g. if $\lambda x : A.\ b \Rightarrow^* c$, then c is syntactically equal to $\lambda x : A'.\ b'$ for some A', b' such that $A \Rightarrow^* A', b \Rightarrow^* b'$.

Proving that conversion is transitive requires proving confluence for parallel reduction. We use the notion of complete development $\boxed{a^{\mathsf{T}}}$ by Takahashi [21], which joins parallel reduction and proves the diamond property. Its definition is omitted here, but corresponds to simultaneous reduction of all redexes.

▶ **Lemma 5** (Completion (p.r.)). ⟨reduction.lean:parTaka⟩ *If $a \Rightarrow b$, then $b \Rightarrow a^{\mathsf{T}}$.*

▶ **Corollary 6** (Diamond (p.r.)). ⟨reduction.lean:diamond⟩ *If $a \Rightarrow b$ and $a \Rightarrow c$, then there exists some d such that $b \Rightarrow d$ and $c \Rightarrow d$. In particular, d is a^{T}, with the reductions given by Completion (p.r.).*

▶ **Theorem 7** (Confluence (p.r.)). ⟨reduction.lean:confluence⟩
If $a \Rightarrow^* b$ and $a \Rightarrow^* c$, then there exists some d such that $b \Rightarrow^* d$ and $c \Rightarrow^* d$.

▶ **Corollary 8** (Properties of conversion). ⟨reduction.lean:conv*⟩ *Conversion is reflexive, symmetric, transitive, substitutive, and congruent. Transitivity requires Confluence (p.r.); the remaining are straightforward from the corresponding properties of parallel reduction.*

Inversion on parallel reduction gives syntactic consistency and injectivity of conversion. Finally, definitional equality is equivalent to conversion, which allows us to use them interchangeably later on.

▶ **Lemma 9** (Syntactic consistency). ⟨reduction.lean:conv{$\mathcal{U}$,Pi,Mty,Lvl}{$\mathcal{U}$,Pi,Mty,Lvl}⟩
If v_1 and v_2 have different syntactic shapes, then $v_1 \Leftrightarrow v_2$ is impossible.

▶ **Lemma 10** (Injectivity (conv.)). ⟨reduction.lean:conv{Pi,$\mathcal{U}$,Lvl}Inv⟩
1. *If $\Pi x : A_1.\ B_1 \Leftrightarrow \Pi x : A_2.\ B_2$, then $A_1 \Leftrightarrow A_2$ and $B_1 \Leftrightarrow B_2$.*
2. *If $\mathsf{U}\ k_1 \Leftrightarrow \mathsf{U}\ k_2$, then $k_1 \Leftrightarrow k_2$.*
3. *If $\mathsf{Level{<}}\ k_1 \Leftrightarrow \mathsf{Level{<}}\ k_2$, then $k_1 \Leftrightarrow k_2$.*

▶ **Theorem 11.** ⟨typing.lean:convEqv,eqvConv⟩ *$a \equiv b$ iff $a \Leftrightarrow b$.*

4.2 Subject reduction and type safety

To prove subject reduction, we need the usual weakening, substitution, replacement, and regularity lemmas. They follow from stronger forms of these lemmas involving simultaneous renaming and substitution, whose details we omit.

▶ **Lemma 12.** ⟨safety.lean:wtWeaken,wtSubst,wtReplace,wtRegularity⟩

- *Weakening. If* $\vdash \Gamma$, $\Gamma \vdash B : \mathsf{U}\ k$, *and* $\Gamma \vdash a : A$, *then* $\Gamma, x : B \vdash a : A$, *where* x *not in* a, A.
- *Substitution. If* $\Gamma \vdash b : B$ *and* $\Gamma, x : B \vdash a : A$, *then* $\Gamma \vdash a[x \mapsto b] : A[x \mapsto b]$.
- *Replacement. If* $A \equiv B$, $\Gamma \vdash B : \mathsf{U}\ k$, *and* $\Gamma, x : A \vdash c : C$, *then* $\Gamma, x : B \vdash c : C$.
- *Regularity. If* $\Gamma \vdash a : A$, *then there exists some* k *such that* $\Gamma \vdash A : \mathsf{U}\ k$.

▶ **Theorem 13** (Subject reduction). ⟨safety.lean:wtPar⟩
If $a \Rightarrow b$ *and* $\Gamma \vdash a : A$, *then* $\Gamma \vdash b : A$.

Proof. By induction on the typing derivation of a. The most complex case is when the reduction is P-Beta, requiring Corollary 8 and Lemma 12. Even so, the proof is standard, and the cases for the universe and level rules in Figure 3 follow from the induction hypotheses. ◀

At this point, we are able to prove admissibility of rule Lam without its first premise, which depends only on regularity.

▶ **Corollary 14** (Lam'). ⟨safety.lean:wtfAbs⟩
Given $\Gamma \vdash \Pi x : A.\ B : \mathsf{U}\ k$ *and* $\Gamma, x : A \vdash b : B$, *we have* $\Gamma \vdash \lambda x : A.\ b : \Pi x : A.\ B$.

For progress and type safety, our notion of evaluation is the reflexive, transitive closure $\boxed{a \rightsquigarrow^* b}$ of call-by-name (cbn) reduction $\boxed{a \rightsquigarrow b}$, which reduces β-redexes and head positions. A single step of cbn reduction embeds into a single step of parallel reduction by induction, which allows us to use Subject reduction. These proofs are also standard.

▶ **Lemma 15** (Progress). ⟨safety.lean:wtProgress⟩
If $\cdot \vdash a : A$, *then either* a *is a value, or* $a \rightsquigarrow b$ *for some* b.

▶ **Theorem 16** (Type safety). ⟨safety.lean:wtSafety⟩
If $\cdot \vdash a : A$ *and* $a \rightsquigarrow^* b$, *then either* b *is a value, or* $b \rightsquigarrow c$ *for some* c.

5 Consistency and canonicity

To prove consistency and canonicity, we use a logical relation to semantically interpret closed types as sets of closed terms; these sets are backward closed under reduction, so if a term reduces to something in the set, then it is also in the set. The empty type is interpreted as the empty set, universes as sets of terms that reduce to types, and level types as sets of terms that reduce to concrete levels. Consistency and canonicity then follow from the fundamental soundness theorem, which states that if a term a has type A, then a is in the interpretation of A. For instance, there is no closed term of the empty type, since it must belong to its interpretation as an empty set, which is a contradiction. The structure of the logical relation and the soundness proof is adapted from the mechanization by Liu [15]. We cover some details here, especially as they pertain to universes and levels.

$$\text{I-Mty}\ \frac{}{[\![\bot]\!]_i \searrow \varnothing} \qquad \text{I-Univ}\ \frac{j < i}{[\![\mathsf{U}\ j]\!]_i \searrow \{z \mid \exists P.\, [\![z]\!]_j \searrow P\}} \qquad \text{I-Level<}\ \frac{}{[\![\mathsf{Level<}\ j_1]\!]_i \searrow \{z \mid \exists j_2.\, z \Rightarrow^* j_2 \wedge j_2 < j_1\}}$$

$$\text{I-Step}\ \frac{A \Rightarrow B \qquad [\![B]\!]_i \searrow P}{[\![A]\!]_i \searrow P} \qquad \text{I-Pi}\ \frac{\begin{array}{c}[\![A]\!]_i \searrow P_1 \qquad \forall y.\, y \in P_1 \to \exists P_2.\, R(y, P_2) \\ \forall y.\, \forall P_2.\, R(y, P_2) \to [\![B[x \mapsto y]]\!]_i \searrow P_2\end{array}}{[\![\Pi x : A.\, B]\!]_i \searrow \{f \mid \forall y.\, \forall P_2.\, R(y, P_2) \to y \in P_1 \to f\ y \in P_2\}}$$

Figure 5 Logical relation for closed types ⟨semantics.lean:Interps⟩

5.1 Logical relation for closed types

The logical relation is written as $\boxed{[\![A]\!]_i \searrow P}$, where A is the type, P is the set of terms, and i is the universe level of the type. A set of terms P is mechanized as a predicate on terms, though we to write $a \in P$ in lieu of $P(a)$ to say that a is in the set, and we use set-builder notation in lieu of explicit abstractions. When proving properties of the logical relation, we require no other axioms than predicate extensionality, which follows from function and propositional extensionality; we explicitly mark the lemmas in which they are used with †.

Because universes are interpreted as sets of types which themselves have interpretations at a lower universe level, to ensure that the interpretation is well defined, the mechanization implements it as an inductive definition parametrized by interpretations at lower levels, then ties the knot by well-founded induction on levels. For clarity and concision, we ignore these details and present the logical relation in Figure 5 without worrying about well-foundedness.

Let us get the easier cases out of the way. The interpretation of the empty type as the empty set is given by rule I-Mty. Rule I-Step backward closes the interpretation under reduction of the type, so a type has an interpretation if it reduces to a type with an interpretation. We show shortly that forward closure under reduction of the type also holds, as well as backward closure under reduction of the *terms* in the interpretations.[1]

Because we consider the interpretation of closed types only, and we have a constructor for backward closure, the only other constructors we need are those for normal, closed types. In particular, we need only consider $\mathsf{U}\ j$ and $\mathsf{Level<}\ j_1$ with concrete levels rather than arbitrary level terms. The interpretation of $\mathsf{Level<}\ j_1$ given by rule I-Level< is the set of level terms strictly less than j_1; more precisely, it is the set of terms that reduce to such concrete levels. The interpretation of $\mathsf{U}\ j$ given by rule I-Univ is the set of types that have an interpretation.

The intuition behind rule I-Pi for function types is that a function f is in its interpretation if for every argument y in the interpretation of the domain, the application $f\ y$ is in the interpretation of the codomain. Because we are dealing with dependent types, the interpretation of the codomain varies with the argument, so we need to ensure first that the interpretation exists for *every* argument in the interpretation of the domain, and that $f\ y$ is in the *particular* interpretation of the codomain. It then sounds like we would want rule I-Pi' below ⟨semantics.lean:interpsPi⟩.

$$\frac{[\![A]\!]_i \searrow P_1 \qquad \forall y.\, y \in P_1 \to \exists P_2.\, [\![B[x \mapsto y]]\!]_i \searrow P_2}{[\![\Pi x : A.\, B]\!]_i \searrow \{f \mid \forall y.\, \forall P_2.\, ([\![B[x \mapsto y]]\!]_i \searrow P_2) \to y \in P_1 \to f\ y \in P_2\}}\ \text{I-Pi'}$$

The problem is that the interpretation is not strictly positive in the conclusion, so I-Pi' as a constructor is not well defined. Rule I-Pi therefore uses an auxiliary relation R that

[1] We do not require forward closure.

relates the argument y to the interpretation of the codomain $B[x \mapsto y]$. Rule I-Pi' then holds by instantiating $R(y, P_2)$ with $[\![B[x \mapsto y]]\!]_i \searrow P_2$ in rule I-Pi. This is the same trick used by Liu [15], whose origins are documented by Anand and Rahli [2].

We require of the logical relation inversion properties for each constructor, along with properties that hold *a priori* for syntactic typing: conversion and cumulativity. A key intermediate lemma is functionality, *i.e.* that the interpretation of a type is deterministic. Cumulativity holds directly by induction on the logical relation. To prove conversion, we begin with closures over reductions.

▶ **Lemma 17** (Cumulativity (l.r.)). ⟨semantics.lean:interpsCumul⟩
Suppose $i < j$. If $[\![A]\!]_i \searrow P$, then $[\![A]\!]_j \searrow P$.

▶ **Lemma 18** (Forward and backward closure (l.r.)). ⟨semantics.lean:interps{Fwds,Bwds}⟩
1. *If $[\![A]\!]_i \searrow P$ and either $A \Rightarrow B$ or $A \Rightarrow^* B$, then $[\![B]\!]_i \searrow P$.*
2. *If $[\![B]\!]_i \searrow P$ and either $A \Rightarrow B$ or $A \Rightarrow^* B$, then $[\![A]\!]_i \searrow P$.*

Proof.
1. For $A \Rightarrow B$, by induction on the logical relation, using Diamond (p.r.) in the I-Step case. Substitution (p.r.) is needed in the I-Pi case to manipulate the substitution in the function codomain. For $A \Rightarrow^* B$, by induction on this reduction.
2. For $A \Rightarrow B$, directly by rule I-Step. For $A \Rightarrow^* B$, by induction on this reduction. ◀

▶ **Corollary 19** (Conversion (l.r.)). ⟨semantics.lean:interpsConv⟩
If $[\![A]\!]_i \searrow P$ and $A \Leftrightarrow B$, then $[\![B]\!]_i \searrow P$, using forward and backward closure.

The final closure lemma we need is backward closure of the terms in the interpretations. When proving the fundamental theorem, we encounter situations where our goal requires inclusion of a reduced term in an interpretation, while induction hypotheses only piece together inclusion of the term before reduction.

▶ **Lemma 20** (Backward closure). ⟨semantics.lean:interpsBwdsP⟩
If $[\![A]\!]_i \searrow P$ and $a \Rightarrow^* b$, then $b \in P$ implies $a \in P$.

Proof. By induction on the logical relation. In the I-Univ case, where a and b are types, we use backward closure from Lemma 18. ◀

The inversion principles for each constructor of the logical relation hold by induction, using properties of parallel reduction as needed. However, it is the inversion principle for rule I-Pi' that we want. The issue lies in the set of terms of the interpretation: if we do not yet know that the sets are unique, then inversion on rule I-Pi gives *some* interpretation P_2 of the codomain, but we do not know whether it is *the* interpretation that is required. We solve this by proving functionality.

▶ **Lemma 21** (Fixed-level functionality (l.r.)). † ⟨semantics.lean:interpsDet'⟩
If $[\![A]\!]_i \searrow P$ and $[\![A]\!]_i \searrow Q$, then $P = Q$.

Proof. By induction on the first logical relation, then generally inversion on the second, except for the I-Step case, which holds directly by the induction hypothesis and forward closure on the second logical relation. The complex case is I-Pi, where we must prove the two sets of terms equal, knowing by the induction hypotheses that the interpretations of the domain and codomain yield equal sets. Because sets are encoded as predicates, we need to use predicate extensionality. It then suffices to show that membership in one set implies membership in the other, which holds using the induction hypotheses. ◀

Functionality holds even with different universe levels, the idea being that the interpretation of a type is independent of the level at which it lives. We are then finally able to prove the inversion property for rule I-Pi'.

▶ **Lemma 22** (Functionality (l.r.)). ⟨semantics.lean:interpsDet⟩
If $[\![A]\!]_i \searrow P$ *and* $[\![A]\!]_j \searrow Q$*, then* $P = Q$*.*

Proof. By totality of the order on levels, either i and j are equal, or one is strictly larger than the other. In the latter case, we use Cumulativity (l.r.) to lift the logical relation at the lower level to the higher level. Then the sets are equal by Fixed-level functionality (l.r.). ◀

▶ **Lemma 23** (Inversion on function types (l.r.)). † ⟨semantics.lean:interpsPiInv⟩
If $[\![\Pi x : A.\, B]\!]_i \searrow P$*, then there exists a* P_1 *such that:*
1. $[\![A]\!]_i \searrow P_1$*;*
2. $\forall y.\, y \in P_1 \to \exists P_2.\, [\![B[x \mapsto y]]\!]_i \searrow P_2$*; and*
3. $P = \{f \mid \forall y.\, \forall P_2.\, ([\![B[x \mapsto y]]\!]_i \searrow P_2) \to y \in P_1 \to f\ y \in P_2\}$.

Proof. By inversion on the logical relation, which gives P_1 and R such that:
4. $[\![A]\!]_i \searrow P_1$;
5. $\forall y.\, y \in P_1 \to \exists P_2.\, R(y, P_2)$;
6. $\forall y.\, \forall P_2.\, R(y, P_2) \to [\![B[x \mapsto y]]\!]_i \searrow P_2$; and
7. $P = \{f \mid \forall y.\, \forall P_2.\, R(y, P_2) \to y \in P_1 \to f\ y \in P_2\}$.

1 holds directly by 4, and 2 holds by combining 5 and 6. To show that the sets in 3 and 7 are equal, we again use predicate extensionality.

- *3 implies 7.* Supposing y and P_2, we have three hypotheses $([\![B[x \mapsto y]]\!]_i \searrow P_2) \to y \in P_1 \to f\ y \in P_2$, $R(y, P_2)$, and $y \in P_1$. From 6 on the second hypothesis, we have $[\![B[x \mapsto y]]\!]_i \searrow P_2$, so we can apply the first hypothesis to get $f\ y \in P_2$.
- *7 implies 3.* Supposing y and P_2, we have three hypotheses $R(y, P_2) \to y \in P_1 \to f\, y \in P_2$, $[\![B[x \mapsto y]]\!]_i \searrow P_2$, and $y \in P_1$. By the first hypothesis on the second and on 5, there exists a P_2' such that $f\ y \in P_2'$. From 6, we also have $[\![B[x \mapsto y]]\!]_i \searrow P_2'$. Then by Functionality (l.r.), we have $P_2 = P_2'$, so $f\ y \in P_2$. ◀

Inversion principles also hold for the other types by induction on the logical relation.

▶ **Lemma 24** (Inversion on universes (l.r.)). ⟨semantics.lean:interps$\mathcal{U}$Inv⟩
If $[\![\mathsf{U}\ k]\!]_i \searrow P$ *and* $A \in P$*, then there exists* j, Q *such that* $k \Rightarrow^* j$ *and* $[\![A]\!]_j \searrow Q$*.*

▶ **Lemma 25** (Inversion on level types (l.r.)). ⟨semantics.lean:interpLvlInv⟩
If $[\![\mathsf{Level}{<}\ \ell]\!]_i \searrow P$ *and* $k \in P$*, then there exist* $j_2 < j_1$ *such that* $\ell \Rightarrow^* j_1$ *and* $k \Rightarrow^* j_2$*.*

▶ **Lemma 26** (Inversion (l.r.)). ⟨semantics.lean:interpsStepInv⟩
If $[\![C]\!]_i \searrow P$*, then one of the following holds:* $C \Rightarrow^* \bot$*; or*
- *There exist* A *and* B *such that* $C \Rightarrow^* \Pi x : A.\, B$*; or*
- *There exists* i *such that* $C \Rightarrow^* \mathsf{U}\ i$ *or* $C \Rightarrow^* \mathsf{Level}{<}\ i$*.*

5.2 Fundamental soundness theorem

Although the logical relation relates closed types to sets of closed terms, the fundamental theorem is proven over syntactic typing of open terms, so we need a notion of semantic typing that handles closing over the terms in a given typing context with a simultaneous substitution. Semantic typing is then elementhood of a term in the interpretation of its type for any substitution that closes them both.

$$\frac{}{\sigma \vDash \cdot}\ \text{I-Nil} \qquad \frac{\sigma \vDash \Gamma \qquad [\![A[\sigma]]\!]_i \searrow P \qquad a \in P}{\sigma, x \mapsto a \vDash \Gamma, x : A}\ \text{I-Cons}$$

Figure 6 Semantically well-typed substitutions ⟨semantics.lean:semSubst{Nil,Cons}⟩

At this point, referring to simultaneous substitutions is inevitable. We denote them as σ, and write $\sigma, x \mapsto a$ for its extension by a single substitution of x by a. In the mechanization, semantic well-typedness of a substitution $\boxed{\sigma \vDash \Gamma}$ is defined similarly to semantic typing $\boxed{\Gamma \vdash a : A}$, but the admissible rules defined in Figure 6 are more convenient.

▶ **Definition 27.** ⟨semantics.lean:semSubst⟩ *A substitution σ is semantically well typed wrt context Γ iff for every $x : A \in \Gamma$, there exist i, P such that $[\![A[\sigma]]\!]_i \searrow P$ and $x[\sigma] \in P$.*

▶ **Definition 28** (Semantic typing). ⟨semantics.lean:semWt⟩ *A term a is semantically well typed with type A under context Γ, written $\Gamma \vDash a : A$, iff for every σ such that $\sigma \vDash \Gamma$, there exist i, P such that $[\![A[\sigma]]\!]_i \searrow P$ and $a[\sigma] \in P$.*

The fundamental soundness theorem states that syntactic typing implies semantic typing. The cases corresponding to the rules in Figure 2 are routine by construction and inversion of rules I-Pi and I-Mty [15], so we do not cover them all here. Instead, we detail only the I-Lam case to highlight where some of the above lemmas are used, followed by the cases for the rules in Figure 3 that are unique to our system. For concision, we skip steps involving massaging substitutions into the right shape.

▶ **Theorem 29** (Soundness). ⟨soundness.lean:soundness⟩ *If $\Gamma \vdash a : A$, then $\Gamma \vDash a : A$.*

Proof. By induction on the typing derivation. In each case, we suppose that $\sigma \vDash \Gamma$.

- *Rule* Lam. The relevant premises are $\Gamma \vdash \Pi x : A.\, B : \mathsf{U}\ k$ and $\Gamma, x : A \vdash b : B$, concluding with $\Gamma \vdash \lambda x : A.\, b : \Pi x : A.\, B$. By the induction hypothesis on the first premise, Lemma 24, and Lemma 23, we have $[\![A[\sigma]]\!]_i \searrow P_1$, $[\![B[\sigma, x \mapsto a]]\!]_i \searrow P_2$, and $a \in P_1$, where the goal is now to show that $(\lambda x : A.\, b)\ a \in P_2$. By rule I-Cons and the induction hypothesis on the second premise, we have $[\![B[\sigma, x \mapsto a]]\!]_{i'} \searrow P_2'$ and $b[x \mapsto a] \in P_2'$ for some i', P_2'. By Functionality (l.r.), we have that $P_2 = P_2'$. Finally, by Backward closure on rule P-Beta and $b[x \mapsto a] \in P_2$, we obtain $(\lambda x : A.\, b)\ a \in P_2$.
- *Rule* Univ. The premise is $\Gamma \vdash k : \mathsf{Level{<}}\ \ell$, concluding with $\Gamma \vdash \mathsf{U}\ k : \mathsf{U}\ \ell$. By the induction hypothesis and Lemma 25, we have $i_1 < i_2$ such that $k[\sigma] \Rightarrow^* i_1$ and $\ell[\sigma] \Rightarrow^* i_2$. By cofinality, there must exist a j such that $i_2 < j$. The goal is now to show that $[\![\mathsf{U}\ (\ell[\sigma])]\!]_j \searrow \{z \mid \exists P.\, [\![z]\!]_{i_2} \searrow P\}$ and $[\![\mathsf{U}\ (k[\sigma])]\!]_{i_2} \searrow \{z \mid \exists P.\, [\![z]\!]_{i_1} \searrow P\}$. These are both constructed using rule I-Univ and Lemma 18.
- *Rule* Level<. The premises are $\Gamma \vdash \mathsf{U}\, k_1 : \mathsf{U}\, \ell_1$ and $\Gamma \vdash k_0 : \mathsf{Level{<}}\ \ell_0$, concluding with $\Gamma \vdash \mathsf{Level{<}}\ k_0 : \mathsf{U}\, k_1$. By the induction hypothesis on the first premise and Lemma 24, $\mathsf{U}\ (k_1[\sigma])$ has an interpretation as a universe, so it remains to find a P such that $[\![\mathsf{Level{<}}(k_0[\sigma])]\!]_j \searrow P$, where $k_1[\sigma] \Rightarrow^* j$. By the induction on the second premise and Lemma 25, we have $k_0[\sigma] \Rightarrow^* i$ for some i. Then the goal is constructed using rule I-Level< and Lemma 18.
- *Rule* Lvl. Straightforward by construction using rule I-Level<.
- *Rule* Trans. The premises are $\Gamma \vdash k_1 : \mathsf{Level{<}}\ k_2$ and $\Gamma \vdash k_2 : \mathsf{Level{<}}\ k_3$, concluding with $\Gamma \vdash k_1 : \mathsf{Level{<}}\ k_3$. By the induction hypotheses on the two premises and Lemma 25, we know that $k_1[\sigma] \Rightarrow^* i_1$, $k_2[\sigma] \Rightarrow^* i_2$, $k_2[\sigma] \Rightarrow^* i_2'$, and $k_3[\sigma] \Rightarrow^* i_3$ such that $i_1 < i_2$ and $i_2' < i_3$. By Confluence (p.r.) and Syntactic consistency, it must be that $i_2 = i_2'$. From the

second inversion, we already know that $\mathsf{Level<}\,(k_3[\sigma])$ has an interpretation, so it remains to show that $k_1[\sigma] \Rightarrow^* i_1$ and $k_3[\sigma] \Rightarrow^* i_3$ such that $i_1 < i_3$, which holds by transitivity.

- *Rule* CUMUL. The premises are $\Gamma \vdash A : \mathsf{U}\ k$ and $\Gamma \vdash k : \mathsf{Level<}\ \ell$, concluding with $\Gamma \vdash A : \mathsf{U}\ \ell$. By induction on the first premise and Lemma 24, we have some P such that $[\![A[\sigma]]\!]_i \searrow P$ and $k[\sigma] \Rightarrow^* i$. By induction on the second premise and Lemma 25, we have some $i' < j$ such that $k[\sigma] \Rightarrow^* i'$ and $\ell[\sigma] \Rightarrow^* j$. By Confluence (p.r.) and Syntactic consistency, it must be that $i = i'$. By cofinality and Lemma 18, $\mathsf{U}\,(\ell[\sigma])$ has an interpretation as a universe. It remains to show that $[\![A[\sigma]]\!]_j \searrow P$, which holds by Cumulativity (l.r.) on $i < j$. ◀

Consistency and canonicity results then follow from the fundamental theorem as corollaries.

▶ **Corollary 30** (Consistency). ⟨soundness.lean:consistency⟩ *There is no b such that $\cdot \vdash b : \bot$ holds. If there were, by Soundness, we get have $\cdot \vDash b : \bot$. Instantiating with the identity substitution, then inverting on the interpretation of $\bot$, we get $b \in \varnothing$, which is a contradiction.*

▶ **Corollary 31** (Canonicity of types). ⟨soundness.lean:canon$\mathcal{U}$⟩ *If $\cdot \vdash C : \mathsf{U}\ k$, then either $C \Rightarrow^* \Pi x : A.\, B$, $C \Rightarrow^* \mathsf{U}\ i$, $C \Rightarrow^* \mathsf{Level<}\ i$, or $C \Rightarrow^* \bot$. By Soundness, instantiating with the identity substitution, we have j, Q such that $[\![\mathsf{U}\ k]\!]_j \searrow Q$ and $C \in Q$. By inversion on the former, we have i, P such that $k \Rightarrow^* i$ and $[\![C]\!]_i \searrow P$. Then the goal holds by Inversion (l.r.).*

▶ **Corollary 32** (Canonicity of levels). ⟨soundness.lean:canonLvl⟩ *If $\cdot \vdash k : \mathsf{Level<}\ \ell$, then $k \Rightarrow^* i$ for some concrete level i. By Soundness, instantiating with the identity substitution, we have j, P such that $[\![\mathsf{Level<}\ \ell]\!]_j \searrow P$ and $k \in P$. By inversion on the former, we have that $\ell \Rightarrow^* i_2$ and $k \Rightarrow^* i_1$ such that $i_1 < i_2$.*

6 Towards normalization

One conventional way to prove normalization, given that we already have a syntactic logical relation, is to extend it from closed to open types and terms. However, we have not yet found the correct interpretation for open universe types that continues to satisfy the same properties we need (inversion, conversion, cumulativity, functionality) while being strong enough for the soundness proof to go through.

It is also unclear whether the issue is finding the correct semantic model, or if normalization does not hold at all, because it depends on the syntactic presentation: if we remove type annotations from our type theory and present it Curry-style, is not normalizing. While directly declaring an ill-founded level $x : \mathsf{Level<}\ x$ is impossible, we can construct such a level in an inconsistent context using an unannotated absurd eliminator. Then it becomes possible to type the universe at this level as its own type. Figure 7 explicitly constructs the key part of the typing derivation for $\mathsf{U}\,(\mathsf{absurd}\ x) : \mathsf{U}\,(\mathsf{absurd}\ x)$ where $x : \bot$. With an instance of type-in-type, we can construct a nonnormalizing lambda term via *e.g.* Hurkens' paradox [13].

The ability to assign different types to the term $\mathsf{absurd}\ x$ is key to constructing this derivation. By requiring a type annotation that gets compared during definitional equality, we can only construct a derivation for $\mathsf{U}\,(\mathsf{absurd}_{(\mathsf{Level<}\,(\mathsf{absurd}_{(\mathsf{Level<}\,0)}\ x))}\ x) : \mathsf{U}\,(\mathsf{absurd}_{(\mathsf{Level<}\,0)}\ x)$, which cannot be used as type-in-type. For similar reasons, we cannot use $x : \Pi A : \mathsf{U}\ i.\, A$ to construct the ill-founded level, as the type arguments will be incomparable. In contrast, type annotations have no influence on consistency, as it remains provable via the logical relation on closed types even when annotations are removed.

$$\dfrac{\dfrac{\dfrac{0 < 1}{x : \bot \vdash 0 : \mathsf{Level}\!<\! 1}\,\textsc{Lvl}}{x : \bot \vdash \mathsf{U}\,0 : \mathsf{U}\,1}\,\textsc{Univ}}{\dfrac{x : \bot \vdash \mathsf{Level}\!<\! (\mathsf{absurd}\ x) : \mathsf{U}\,0}{x : \bot \vdash \mathsf{absurd}\ x : \mathsf{Level}\!<\! (\mathsf{absurd}\ x)}\,\textsc{Abs}}{x : \bot \vdash \mathsf{U}\,(\mathsf{absurd}\ x) : \mathsf{U}\,(\mathsf{absurd}\ x)}\,\textsc{Univ}$$

$$\dfrac{\dfrac{\dots}{x : \bot \vdash \mathsf{Level}\!<\! 0 : \mathsf{U}\,0}\qquad \dfrac{x : \bot \in x : \bot}{x : \bot \vdash x : \bot}\,\textsc{Var}}{x : \bot \vdash \mathsf{absurd}\ x : \mathsf{Level}\!<\! 0}\,\textsc{Abs}$$

$$\dfrac{x : \bot \in x : \bot}{x : \bot \vdash x : \bot}\,\textsc{Var}$$

Figure 7 Type-in-type in an inconsistent context

7 Extensions

Our type theory is intentionally minimal to focus only on the core necessities of first-class levels and to keep the proof development small and uncluttered. Some reasonable extensions include the remaining missing types from MLTT, *i.e.* dependent pairs, sums, naturals, propositional equality, and W types, or general inductive types as in CIC [22]. However, these features and their difficulties are orthogonal from universes and levels. Here, we instead look at extensions that augment how universes and levels behave, some of which are validated by our current semantics, and others which present additional challenges.

7.1 Level operators and eliminators

The only features missing from TTBFL that Agda has are a zeroth level, a level successor operator, and a level maximum operator. To justify them semantically, we would impose the first two as additional existence conditions on the metalevel levels; the third follows from the total ordering, which lets us pick the larger of two levels.

$$\textsc{Zero}\ \dfrac{\Gamma \vdash k : \mathsf{Level}\!<\! \ell}{\Gamma \vdash 0 : \mathsf{Level}\!<\! (\uparrow k)} \qquad \textsc{Succ}\ \dfrac{\Gamma \vdash k : \mathsf{Level}\!<\! \ell}{\Gamma \vdash \uparrow k : \mathsf{Level}\!<\! (\uparrow \ell)} \qquad \textsc{Max}\ \dfrac{\Gamma \vdash k_1 : \mathsf{Level}\!<\! \ell_1 \qquad \Gamma \vdash k_2 : \mathsf{Level}\!<\! \ell_2}{\Gamma \vdash k_1 \sqcup k_2 : \mathsf{Level}\!<\! (\ell_1 \sqcup \ell_2)}$$

What complicates matters are the additional definitional equalities that ensure that the maximum operator is idempotent, associative, commutative, distributive with respect to successors, and that 0 is its identity element. While these properties hold automatically at the metalevel for concrete levels, they do not for arbitrary level expressions, *e.g.* $0 \sqcup \uparrow (x \sqcup \uparrow x) \equiv \uparrow\uparrow x$. Our notions of reduction then need to pick a direction for each equality to reduce levels to some chosen canonical form. We believe the mechanization to be doable but tedious.

Meanwhile, well-founded induction on levels already holds semantically, as we need it to define our logical relation in the first place. We can internalize it by syntactically introducing an eliminator wf for levels, which states that a predicate B holds on arbitrary levels if we can show that it holds for a given level when we know it holds for all smaller levels. However, it is unclear whether such an eliminator would be useful.

$$\textsc{ElimLvl}\ \dfrac{\Gamma, z : \mathsf{Level}\!<\! k \vdash B : \mathsf{U}\,\ell \qquad \Gamma \vdash b : \Pi x : \mathsf{Level}\!<\! k.\, (\Pi y : \mathsf{Level}\!<\! x.\, B[z \mapsto y]) \to B[z \mapsto x]}{\Gamma \vdash \mathsf{wf}\ b : \Pi z : \mathsf{Level}\!<\! k.\, B} \qquad \textsc{E-ElimLvl}\ \dfrac{}{\mathsf{wf}\ b\ k \equiv b\ k\ (\lambda y.\, \mathsf{wf}\ b\ y)}$$

7.2 Subtyping

Because levels are now terms, subtyping necessarily involves typing to compare two levels. In particular, a universe at a smaller level is a subtype of a one at a larger level, while a level type bounded by a smaller level is a subtype of a one bounded by a larger level. The former is already expressed by rule Cumul, the latter by rule Trans. The additional benefit of subtyping making function domains contravariant and codomains covariant with respect to subtyping. Selected subtyping rules are given below, along with an updated rule Conv' rule.

$$\text{S-Univ}\ \frac{\Gamma \vdash k : \mathsf{Level}\!<\ell}{\Gamma \vdash \mathsf{U}\ k \preccurlyeq \mathsf{U}\ \ell} \qquad \text{S-Level}< \frac{\Gamma \vdash k : \mathsf{Level}\!<\ell}{\Gamma \vdash \mathsf{Level}\!<\ k \preccurlyeq \mathsf{Level}\!<\ \ell} \qquad \text{S-Pi}\ \frac{\Gamma \vdash A_2 \preccurlyeq A_1 \qquad \Gamma, x : A_1 \vdash B_1 \preccurlyeq B_2}{\Gamma \vdash \Pi x : A_1.\ B_1 \preccurlyeq \Pi x : A_2.\ B_2}$$

$$\text{S-Trans}\ \frac{\Gamma \vdash A \preccurlyeq B \qquad \Gamma \vdash B \preccurlyeq C}{\Gamma \vdash A \preccurlyeq C} \qquad \text{S-Conv}\ \frac{A \equiv B}{\Gamma \vdash A \preccurlyeq B} \qquad \text{Conv'}\ \frac{\Gamma \vdash a : A \qquad \Gamma \vdash B : \mathsf{U}\ k \qquad \Gamma \vdash A \preccurlyeq B}{\Gamma \vdash a : B}$$

Although all of this subtyping behaviour holds semantically in our current model, proving logical consistency is not so easy. The simplicity of our logical relation relies on the independence of definitional equality from typing, along with its equivalence to conversion. By introducing a subtyping judgement that depends on typing, which in turn depends on subtyping, to prove consistency, the logical relation would need to include a semantic notion of equality, similar to the reducibility judgements used by Abel, Öhman, and Vezzosi [1].

8 Conclusion and future work

We have presented TTBFL, a type theory with first-class universe levels. In contrast to existing work, rather than level constraints being separate from the type of levels, we combine them such that every level explicitly has a bound. We have proven our type theory to be type safe, and in particular that subject reduction holds. This is in contrast to BCDE [3], the only other formal syntactic system we know of with universe level polymorphism beyond prenex polymorphism, which violates subject reduction. We have also proven our type theory to be logically consistent, and therefore useable as a logic for writing proofs.

Proving normalization and decidability of type checking is the next step in showing that our type theory is effectively type checkable and thus has the potential to be a basis for theorem proving. Whether the extended logical relation presented in Section 6 can be repaired to prove normalization is unclear, as is whether well-typed terms are normalizing at all. Looking to existing work, BCDE proposes allowing looping level constraints of the form $k < k$ to admit subject reduction, but this would also permit type-in-type in a looping context and violate normalization. Even so, we are hopeful that it holds, as no issues with cumulative first-class levels have yet arisen in Agda.

Decidability of type checking does not hold straightforwardly from normalization, as a type checking algorithm must incorporate the non–syntax-directed rules Trans and Cumul. It may be done separately via algorithmic subtyping, but as seen in Section 7.2, a subtyping relation must depend on typing to show that one level expression is strictly smaller than another. The challenge lies in showing totality of a mutual typing–subtyping algorithm, but if looping level bounds $k : \mathsf{Level}\!<\ k$ are ruled out by normalization, there is no reason to believe it would not be total.

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. Abhishek Anand and Vincent Rahli. Towards a formally verified proof assistant. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, pages 27–44, Cham, 2014. Springer International Publishing.
3. Marc Bezem, Thierry Coquand, Peter Dybjer, and Martín Escardó. Type Theory with Explicit Universe Polymorphism. In Delia Kesner and Pierre-Marie Pédrot, editors, *28th International Conference on Types for Proofs and Programs (TYPES 2022)*, volume 269 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:16, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://arxiv.org/abs/2212.03284, doi:10.4230/LIPIcs.TYPES.2022.13.
4. The Coq Development Team. The coq proof assistant, September 2024. URL: https://coq.github.io/doc/v8.20/refman, doi:10.5281/zenodo.14542673.
5. Judicaël Courant. Explicit Universes for the Calculus of Constructions. In Victor A. Carreño, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 115–130, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.
6. Menno de Boer. A Proof and Formalization of the Initiality Conjecture of Dependent Type Theory, 2020.
7. 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.
8. 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. Series Title: Lecture Notes in Computer Science. doi:10.1007/3-540-61780-9_66.
9. Lucas Escot and Jesper Cockx. Practical generic programming over a universe of native datatypes. *Proc. ACM Program. Lang.*, 6(ICFP), August 2022. doi:10.1145/3547644.
10. Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD dissertation, Université Paris VII, 1972.
11. Robert Harper and Robert Pollack. Type checking with universes. *Theoretical Computer Science*, 89(1):107–136, Oct 1991. URL: https://linkinghub.elsevier.com/retrieve/pii/030439759090108T, doi:10.1016/0304-3975(90)90108-T.
12. 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.
13. Antonius J. C. Hurkens. A simplification of Girard's paradox. In *Typed Lambda Calculi and Applications*, pages 266–278, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. doi:10.1007/BFb0014058.
14. 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://arxiv.org/abs/2103.00223, doi:10.4230/LIPIcs.CSL.2022.28.
15. Yiyun Liu, Jonathan Chan, and Stephanie Weirich. Functional Pearl: Short and Mechanized Logical Relation for Dependent Type Theories, 2025. Proof pearl under submission. URL: https://github.com/yiyunliu/mltt-consistency/.
16. Per Martin-Löf. An intuitionistic theory of types: predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic colloquium '73*, Studies in logic and the foundations of mathematics, pages 73–118. North-Holland Publishing Company, Amsterdam and Oxford, and American Elsevier Publishing Company, July 1975.

17 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.

18 Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Botsch Nielsen, Nicolas Tabareau, and Théo Winterhalter. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. *Journal of the ACM (JACM)*, pages 1–76, November 2024. URL: https://inria.hal.science/hal-04077552, doi:10.1145/3706056.

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

20 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.

21 Masako Takahashi. Parallel Reductions in λ-Calculus. *Information and Computation*, 118(1):120–127, 1995. doi:10.1006/inco.1995.1057.

22 Amin Timany and Matthieu Sozeau. Cumulative Inductive Types In Coq. In Hélène Kirchner, editor, *3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018)*, volume 108 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.FSCD.2018.29.