

The con-leche proof idea

Joachim Breitner, Lean FRO

Revision 05c734ec21, 2026-10-14

This document was written by an AI agent (Claude, working with the maintainer). It is a self-contained account of the core proof idea of con-leche, a verified kernel checker for Lean 4: the modelling and consistency argument that connects Lean’s own terms, an inductive description of checking, and a set theory given axiomatically. The real proof lives in the repository, and phrases underlined like this one link to it on `master`, at the definition or theorem being described; phrases underlined like this one link to the paper’s own Lean fragment. In the web version, hovering over such a phrase shows the cited lines.

Contents

1. Introduction	2
1.1. What is proved	2
1.2. The shape of the argument	2
1.3. How to interpret a $\forall$ or a lambda	2
1.4. This document	3
2. The fragment without an environment	3
2.1. Terms	3
2.2. Levels	4
2.3. The three relations	4
2.4. The set theory we assume	7
2.5. The interpretation	8
2.6. The semantic invariant	9
2.7. The three claims and their proof	10
2.8. The certification tax	13
2.9. The cheaper mode of inference	14
2.10. The argument in brief	14
3. Adding definitions	15
4. Adding inductive types	17
4.1. What is checked	17
4.2. The model	20
4.3. Consistency	28
5. Nested inductive types	28
5.1. Nested blocks	29
5.2. The positivity check	29
5.3. Classes and their recursors	31
5.4. The model, as a sketch	32
6. Consequences of extensionality	33
6.1. Proof irrelevance and propositional extensionality	33
6.2. η for structures	33
6.3. Unit-likeness	34
7. What we left out	35
7.1. Dropped features	35
7.2. How the real proof differs from the fragment	36
8. Related work	37
References	39

1. Introduction

1.1. What is proved

con-leche is a checker for Lean 4. It reads an export of a Lean environment, the *stream*: every definition, theorem and inductive type, in the official kernel’s own terms. It accepts or rejects the stream, checking what the official kernel checks (on a few features it declines instead; Section 7). It comes with a proof, written in Lean itself, of what an acceptance means: every environment the checker accepts has a model in set theory. Each constant is assigned a set, each type denotes a set, and every stored constant is a member of the set its type denotes. The constant `False` denotes the empty set, and `Eq` denotes set equality, so every equation the checker accepts is an equality of sets. Hence no accepted environment holds a proof of `False`, and every accepted theorem is true in the model.

1.2. The shape of the argument

The theorem is relative to a model of an abstract set theory, a structure with membership, extensionality, the usual set-forming operations and a chain of universes closed under them. By Gödel’s second incompleteness theorem, Lean cannot prove that such a structure exists. The assumption is standard, though: it follows from the existence of ω many inaccessible cardinals, the hypothesis of the Lean development of Carneiro’s model of Lean (Section 8; §7 of [OVERVIEW.md](#) says more).

The usual way to prove such a statement, given a typing judgement for the type theory, has two steps: show that the checker accepts only derivable judgements, and show that derivable judgements are true in the model. The first step needs the metatheory of the type theory: that reduction preserves types (subject reduction), that reduction is confluent, that a function type determines its domain and codomain (injectivity of Π), and that a term has one type up to definitional equality. Our proof needs none of it.

Nor does it have a typing judgement. In its place is a description of what the checker *does*: three inductively defined relations, one for reduction, one for the verdicts of the definitional-equality test and one for type inference (six in the real proof, where premises about lists get relations of their own; Section 7). Their rules are exactly the moves the checker makes, and each rule’s premises are what the checker verified at that point. On the model side there is a total, term-directed interpretation: every term denotes a set, well-typed or not, and the interpretation uses nothing beyond what is in the term itself. In particular it uses no typing information, nothing that would have to be inferred. Where a typing judgement would say “this term has that type”, there is a semantic invariant on the term’s set, imposed at every subterm: every application applies a function to a member of its domain, every binder’s domain is a type, and so on. One induction over the three relations then proves three claims at once. A reduction step preserves the denotation and the semantic invariant; a verdict “definitionally equal” means that the two sides denote the same set; and an inferred type contains the term, that is, the term’s set is a member of the type’s set.

1.3. How to interpret a $\forall$ or a lambda

At every binder the interpretation has to decide how to interpret a $\forall$ or a lambda. In Lean, $\forall x : A, B$ is a type of functions when B is a type and a proposition when B is a proposition, and in the model these are different kinds of sets. A function type is a set of graphs. A proposition is a *truth value*, a set with at most one element: the model has one distinguished set, *the point*, which every proof of every proposition denotes, so proof irrelevance is built in. The two cases look the same on the page, and telling them apart by the sort of B would take type inference. Our interpretation instead reads the answer off the values: a $\forall$ denotes a truth value exactly when every one of its fibres (the set B denotes at a member of the domain) is a truth value, and a lambda denotes the point exactly when every value of its body is the point. The terms are Lean’s own.

Over a non-empty domain this reading agrees with the sort of the body. Over an empty domain it does not: `fun h => h.elim`, of type `False → Nat`, denotes the point although its type has sort `Type` (Section 2.5).

This shapes the soundness claims: a reduction or an equality claims nothing about a term that denotes the point (Section 2.7). It also costs two checks that the official kernel never makes (Section 2.8): the checker compares the types of two bodies at lambda congruence and η , and, in one rare case of ι -reduction, it compares a constructor application with the recursor’s arguments and type-checks its fields.

1.4. This document

This document is a pen-and-paper account of that argument on a simplified fragment of the checker. The fragment has no projections, no number or string literals, no mutual inductive types, nesting only one container deep and not under a binder (Section 5 describes the full check, without proof), no quotients, no axioms, no built-in copies of `False`, `Eq`, `Nat` and their kin (the real checker pins these rather than reading them from the stream), no distinction between a theorem and a definition, and none of the checker’s performance devices (memo tables, fuel, the parallel check phase). Section 7 lists every omission with one sentence on what the real proof does about it. The fragment is verified in Lean in a small development of its own, `whitepaper/Fragment/`, which imports nothing from the main proof and is parametric in the same kind of abstract set theory. The underlined source links lead into the real proof and into that development.

The rest is in seven parts. §2 presents the fragment without an environment: terms and universe levels, the three relations, the abstract set theory, the interpretation, the semantic invariant, and the three claims with their proof. The routine cases of the proof are listed briefly; the cases where the reading of a binder matters, or where a syntactic proof would fail, are discussed in full. §2 ends with the checks that other checkers skip and with a cheaper mode of inference. The claims (Section 2.7) are a good place to start reading. §3 adds the environment, the constants that refer to it, and its definitions: what is stored, what is checked, how a definition unfolds, and the model that grows with it. §4 adds inductive types as least fixed points in the model, their recursors and reduction rules, large elimination, and the consistency corollary: no accepted environment stores a constant whose type is an inductive proposition with no constructors. §5 changes tack. It describes in full `con-leche`’s check for nested inductive types, whose constructors mention the type being defined inside another one, as in `List (Tree  $\alpha$ )`, and their generated recursors, and only sketches how the model extends to them. §6 shows that two features of Lean’s definitional equality that the fragment drops, η for structures and unit-likeness, follow from the extensionality of the model, with one proof case each. §7 lists what was left out, and how the real proof differs from the fragment where they share a feature. Section 8 relates the argument to earlier work.

2. The fragment without an environment

This section presents the core of the argument on a fragment of Lean without an environment: first the terms, the levels and the three relations that say what the checker does; then the set theory, the interpretation, the semantic invariant that replaces a typing judgement, and the three soundness claims with their proofs. Everything here is verified in the fragment’s own Lean development and linked, alongside its counterpart in the real proof.

2.1. Terms

The terms of the fragment are the official kernel’s terms, minus what Section 7 leaves out:

$$e ::= x \mid \text{Sort } u \mid e \, e \mid \lambda x : A. e \mid \forall x : A. e \mid \text{let } x : T := v; e$$

A term is a variable x ; a sort `Sort  $u$` at a universe level u (Section 2.2), with `Prop` = `Sort 0`; an application $f \, a$; a function $\lambda x : A. b$; a dependent function type, or Π -type, $\forall x : A. B$, written $A \rightarrow B$ when x does not occur in B ; or a let-expression `let  $x : T := v$ ;  $b$` . Types are terms; there is no separate class. A sequence of binders, such as the lambdas of $\lambda x : A. \lambda y : B. e$, is a *telescope*.

The paper writes named variables, with the usual conventions: terms are taken up to renaming of bound variables, and $B[x := a]$ is the capture-avoiding substitution of a for x . The Lean fragment uses de Bruijn

indices instead (a variable is the number of binders between its occurrence and its own), and a rule that mentions the types of the variables in scope carries a *context* Γ , a list of those types. No rule in this paper needs index arithmetic in its named form.

IN CON-LECHE The checker uses neither device. It opens a binder with a fresh free variable that carries its own type, so it keeps no context at all. This is a performance device and changes nothing below.

2.2. Levels

Universe levels are Lean's, without metavariables:

$$u ::= 0 \mid u + 1 \mid \max(u, v) \mid \text{imax}(u, v) \mid p$$

with p a level parameter of the enclosing declaration. A level means a natural number once a valuation φ fixes the parameters: 0, successor, maximum, and $\text{imax}(u, v)$ is 0 when v is 0 and $\max(u, v)$ otherwise (as in con-leche). The impredicative maximum is what makes $\forall x : A. B$ a proposition whenever B is one, whatever the level of A : its sort is $\text{Sort } (\text{imax}(u, v))$ (rule pi in Section 2.3).

The level oracle. The checker decides $u \leq v$ and $u = v$ between levels with an algorithm. This paper does not present that algorithm and does not verify it. It assumes two Boolean functions with a specification: $u \leq v$ is answered yes exactly when the value of u is at most the value of v at every valuation, and $u = v$ exactly when the values agree at every valuation (the real $\leq$, the real $=$ and their soundness proofs). We write $u \doteq v$ for “the oracle says $u = v$ ”.

2.3. The three relations

The checker is described by three relations over terms, defined together by inference rules: *reduction* $\Gamma \vdash e \rightsquigarrow e'$, *definitional equality* $\Gamma \vdash a \equiv b$, and *inference* $\Gamma \vdash e \Rightarrow T$. A context Γ lists the variables in scope with their types. The complete rule sets are one mutual inductive definition, as in con-leche.

How to read them. The three relations describe what the checker does. $\Gamma \vdash a \equiv b$ means that on the terms a and b the checker's equality test answers yes. $\Gamma \vdash e \Rightarrow T$ means that, asked for the type of e , the checker returns T . We write $\Rightarrow$ rather than the customary colon so that it is not read as a typing judgement, which this paper does not have. Every relation is stated in the accepting direction only; nothing says what the checker rejects. A rule's premises are exactly what the checker did at that site: the recursive calls it made, and the tests whose outcome fixes the shape of the conclusion.

A convention: a chain such as $\Gamma \vdash a \Rightarrow T \rightsquigarrow e$ abbreviates consecutive premises in the same context, one per arrow, the right end of each arrow being the subject of the next; here $\Gamma \vdash a \Rightarrow T$ and $\Gamma \vdash T \rightsquigarrow e$. Chains of any length are read the same way.

2.3.1. Reduction

Reduction chains the head steps the checker makes when it normalises a term. It is reflexive and transitive, and it reaches inside the head of an application.

$$\frac{}{\Gamma \vdash e \rightsquigarrow e} \text{ REFL} \quad \frac{\Gamma \vdash e_1 \rightsquigarrow e_2 \quad \Gamma \vdash e_2 \rightsquigarrow e_3}{\Gamma \vdash e_1 \rightsquigarrow e_3} \text{ TRANS} \quad \frac{\Gamma \vdash f \rightsquigarrow f'}{\Gamma \vdash f a \rightsquigarrow f' a} \text{ HEAD}$$

β fires at every redex, and ζ substitutes a let-binding's value into its body; neither has a premise.

$$\frac{}{\Gamma \vdash (\lambda x : A. b) a \rightsquigarrow b[x := a]} \text{ BETA} \quad \frac{}{\Gamma \vdash (\text{let } x : T := v; b) \rightsquigarrow b[x := v]} \text{ ZETA}$$

As in the official kernel, β has no premise, although its soundness needs the argument to lie in the domain of the lambda; Section 2.6 shows where that comes from. ζ is simpler: a let-expression always denotes what its body with the value substituted denotes.

2.3.2. Definitional equality

The checker's equality test is reflexive and symmetric. The checker never takes a symmetry step; the rule is there so that each one-sided rule below (red-l, η) need be written once, and its mirror image follows. The test interleaves with reduction in one way: reduce the left side, then continue. Sorts are compared through the level oracle.

$$\frac{}{\Gamma \vdash a \equiv a} \text{REFL} \quad \frac{\Gamma \vdash a \equiv b}{\Gamma \vdash b \equiv a} \text{SYM}$$

$$\frac{\Gamma \vdash a \rightsquigarrow a' \quad \Gamma \vdash a' \equiv b}{\Gamma \vdash a \equiv b} \text{RED-L} \quad \frac{u \doteq v}{\Gamma \vdash \text{Sort } u \equiv \text{Sort } v} \text{SORT}$$

The congruences descend into the two binders and into applications. Domains are compared first, then the bodies, under the right-hand domain. For $\forall$ that is all.

$$\frac{\Gamma \vdash A_1 \equiv A_2 \quad \Gamma, x : A_2 \vdash B_1 \equiv B_2}{\Gamma \vdash \forall x : A_1. B_1 \equiv \forall x : A_2. B_2} \text{PI} \quad \frac{\Gamma \vdash f_1 \equiv f_2 \quad \Gamma \vdash a_1 \equiv a_2}{\Gamma \vdash f_1 a_1 \equiv f_2 a_2} \text{APP}$$

Two lambdas are compared as two lambda telescopes, by an auxiliary relation $\Gamma \vdash a \equiv_{\lambda} b$. The checker descends level by level while both bodies are lambdas, comparing the domains at each level (lam-outer). At the innermost level, where the descent stops, it also infers the types of the two bodies under the binder and compares them (lam). That comparison is a check the official kernel does not make. Such checks are typeset in one colour, like this, throughout the paper; a reader who ignores the coloured premises sees the official kernel's rule. Section 2.8 explains why they are there, and why the outer levels need none.

$$\frac{\Gamma \vdash A_1 \equiv A_2 \quad \Gamma, x : A_2 \vdash b_1 \Rightarrow t_1 \quad \Gamma, x : A_2 \vdash b_2 \Rightarrow t_2 \quad \Gamma, x : A_2 \vdash b_1 \equiv b_2 \quad \Gamma, x : A_2 \vdash t_1 \equiv t_2}{\Gamma \vdash \lambda x : A_1. b_1 \equiv_{\lambda} \lambda x : A_2. b_2} \text{LAM}$$

$$\frac{\Gamma \vdash A_1 \equiv A_2 \quad \Gamma, x : A_2 \vdash b_1 \equiv_{\lambda} b_2}{\Gamma \vdash \lambda x : A_1. b_1 \equiv_{\lambda} \lambda x : A_2. b_2} \text{LAM-OUTER} \quad \frac{\Gamma \vdash a \equiv_{\lambda} b}{\Gamma \vdash a \equiv b} \text{LAMS}$$

(Two syntactically equal terms are also related by $\equiv_{\lambda}$.) A lambda against a term b that is not a lambda is tried by η . The type of b is inferred and reduced to a $\forall$, its domain is compared with the lambda's, and the body is compared with b applied to the bound variable. As at the innermost level of lambda congruence, the type of the lambda's body is inferred and compared with the $\forall$'s body. Two terms are equal by proof irrelevance when both are proofs: each one's type has type $\text{Sort } u$ with u oracle-equal to 0.

$$\frac{\Gamma \vdash b \Rightarrow T \rightsquigarrow \forall x : A_2. B \quad \Gamma \vdash A_2 \equiv A_1 \quad \Gamma, x : A_1 \vdash b_1 \equiv b \quad \Gamma, x : A_1 \vdash b_1 \Rightarrow t_1 \quad \Gamma, x : A_1 \vdash t_1 \equiv B}{\Gamma \vdash \lambda x : A_1. b_1 \equiv b} \text{FUN-ETA}$$

$$\frac{\Gamma \vdash a \Rightarrow T_a \Rightarrow S_a \rightsquigarrow \text{Sort } u \quad u \doteq 0 \quad \Gamma \vdash b \Rightarrow T_b \Rightarrow S_b \rightsquigarrow \text{Sort } v \quad v \doteq 0}{\Gamma \vdash a \equiv b} \text{PROOF-IRREL}$$

Proof irrelevance. The rule compares the sorts of the two types, and never the two types. The official kernel also compares T_a with T_b , so that a and b are proofs of the same proposition. This checker does not, and so accepts strictly more than Lean at this one site. The model justifies it: every proof denotes the point, so two proofs of two propositions denote the same set.

The list has no transitivity rule, and none can be added; Section 2.7 explains why. Instead of chaining equalities, the checker chains reductions: it reduces a side, then continues (red-l).

IN CON-LECHE In con-leche a transitivity rule would also combine two rules that are each sound alone into an unsound one. Free variables are compared by index, not by the types they carry, and a fast path equates two

terms whose heads' declared types say they are proofs. Chained, the two would equate any two variables: give each the type of a proposition, then compare the two proofs.

2.3.3. Inference

A variable's type is read off the context, and a sort has the next sort. The rules of this subsection are the checker's *full inference*. The checker also has a cheaper mode, `infer_only`, which skips some of their checks; Section 2.9 presents it.

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A} \text{VAR} \quad \frac{}{\Gamma \vdash \text{Sort } u \Rightarrow \text{Sort } (u + 1)} \text{SORT}$$

The two binder rules are Lean's own. For a $\forall$, the domain's type must reduce to a sort $\text{Sort } u$ and the body's type, under the domain, to a sort $\text{Sort } v$; the result is $\text{Sort } (\text{imax}(u, v))$. For a lambda, the domain's type must reduce to a sort, and the result is the $\forall$ of the body's inferred type; the sort of that type is not computed.

$$\frac{\Gamma \vdash A \Rightarrow S \rightsquigarrow \text{Sort } u \quad \Gamma, x : A \vdash B \Rightarrow T \rightsquigarrow \text{Sort } v}{\Gamma \vdash \forall x : A. B \Rightarrow \text{Sort } (\text{imax}(u, v))} \text{PI}$$

$$\frac{\Gamma \vdash A \Rightarrow S \rightsquigarrow \text{Sort } u \quad \Gamma, x : A \vdash b \Rightarrow B}{\Gamma \vdash \lambda x : A. b \Rightarrow \forall x : A. B} \text{LAM}$$

An application infers the head's type, reduces it to a $\forall$, infers the argument's type and compares it with the domain. A let-expression checks that its type has a sort and that its value's type agrees with it, as Lean does, and then infers the type of the body with the value substituted.

$$\frac{\Gamma \vdash f \Rightarrow T \rightsquigarrow \forall x : A. B \quad \Gamma \vdash a \Rightarrow T_a \quad \Gamma \vdash T_a \equiv A}{\Gamma \vdash f a \Rightarrow B[x := a]} \text{APP}$$

$$\frac{\Gamma \vdash T \Rightarrow S \rightsquigarrow \text{Sort } u \quad \Gamma \vdash v \Rightarrow T_v \quad \Gamma \vdash T_v \equiv T \quad \Gamma \vdash b[x := v] \Rightarrow B}{\Gamma \vdash (\text{let } x : T := v; b) \Rightarrow B} \text{LET}$$

Reducing to a $\forall$. The application rule reduces the inferred type of f until a $\forall$ is syntactically there. It does not take a type of f that is a $\forall$ up to definitional equality and argue that its domain is the right one. That argument is where a syntactic soundness proof needs injectivity of $\forall$, the property that $\forall x : A. B \equiv \forall x : A'. B'$ forces $A \equiv A'$. Our proof never needs it, and the model does not validate it: $\forall x : A. \text{True}$ and $\forall x : A'. \text{True}$ denote the same truth value whatever A and A' are. Our proof argues instead that the head denotes a member of what the $\forall$ denotes, because reduction preserves denotations; that the argument denotes a member of the domain, by the comparison premise; and that applying a member of a function space to a member of its domain lands in the fibre. The second half of this section carries this out.

The let-expression. The two checks on the value are what Lean runs, and they decide verdicts: they reject a let-expression whose value does not have the stated type even where the body ignores it. The soundness proof does not use them (Section 2.7): the interpretation reads no type, and the body is inferred with the value itself in place of the variable.

IN CON-LECHE The checker substitutes the value in the same way, both when it reduces and when it infers; it keeps no variables bound to values. The official kernel opens the body at such a variable instead, which avoids copying the value when the body uses it many times; for con-leche that case is a known performance gap, not a soundness question.

2.4. The set theory we assume

The other side of the argument is a set theory. We do not construct one. The reader is given an axiomatised theory, a type of sets with the operations and laws listed below, and is promised that nothing beyond these laws is used. The theory is stated in Lean’s own logic, so it is higher-order: separation takes any predicate of that logic, and graphs and function spaces are formed from any function of it. It is deliberately not minimal: graphs, dependent function spaces and the universe chain, which a minimal axiom system would construct, are assumed outright with their laws, because how they are built does not matter to the argument. The promise is literal: in the Lean fragment the structure is a class, `SetLib`, and every theorem of the fragment is proved against that class.

IN CON-LECHE The real proof is parametric in a smaller interface, ZF without infinity plus a chain of Grothendieck universes (sets closed under all the set-forming operations), from which it derives the operators below.

The laws come in four groups.

Sets, membership, extensionality, separation. A type V of sets with a membership relation $x \in y$; extensionality: two sets with the same members are equal; and separation: for every set A and every property P of sets, a set $\{x \in A \mid P(x)\}$ whose members are the members of A that satisfy P . A property here is any predicate on sets expressible in Lean’s type theory, the logic the theory is stated in, not only a first-order formula.

The point and the truth values. A distinguished set `pt`, *the point*, and its singleton $\{\text{pt}\}$. The point will be the one proof of every true proposition. For a proposition P of the ambient logic we write $\text{tv}(P)$ for the separation $\{x \in \{\text{pt}\} \mid P\}$ and call it the truth value of P . Its only possible member is the point, which it has exactly when P holds, so $\text{tv}(P) = \{\text{pt}\}$ when P is true and $\text{tv}(P) = \emptyset$ when it is false. This is an abbreviation, not an axiom. The point itself obeys three laws: it is not a truth value (it has a member other than itself), it has at most one member, and no graph (below) is a member of it.

IN CON-LECHE The point is the singleton of the set $\{\emptyset, \{\{\emptyset\}\}\}$, chosen so that no encoding of a value of a data type produces it; the three laws are theorems there.

The universes. A chain of sets $\mathcal{U}_0, \mathcal{U}_1, \mathcal{U}_2, \dots$, with $\mathcal{U}_0$ the set of truth values, the subsets of $\{\text{pt}\}$. Each $\mathcal{U}_n$ is a member of $\mathcal{U}_{n+1}$. The chain is cumulative (a member of $\mathcal{U}_m$ is a member of every later $\mathcal{U}_n$), and every universe is closed under separation.

Graphs, function spaces, application. For a function F from sets to sets (a function of the meta-theory, not a set) and a set A , a set $\text{graph}(F, A)$; for a set A and a family B of sets indexed by sets, again a function of the meta-theory, a set $\Pi(A, B)$, the *dependent function space*; and for two sets f and a a set $f \cdot a$, the application. Their laws:

- $\text{pt} \cdot a = \text{pt}$: a proof applied to anything is the proof.
- β : if $a \in A$ then $\text{graph}(F, A) \cdot a = F(a)$.
- $\text{graph}(F, A) \neq \text{pt}$: a graph is never the point.
- Congruence: $\text{graph}(F, A)$ depends only on the values of F on A , and $\Pi(A, B)$ only on the values of B on A .
- Introduction: if $F(x) \in B(x)$ for every $x \in A$, then $\text{graph}(F, A) \in \Pi(A, B)$.
- Elimination: if $f \in \Pi(A, B)$ and $a \in A$, then $f \cdot a \in B(a)$.
- η : if $f \in \Pi(A, B)$, then $\text{graph}(x \mapsto f \cdot x, A) = f$.
- Domain uniqueness: if $f \in \Pi(A, B)$ and $f \in \Pi(A', B')$, then $A = A'$.
- Closure: for $n \neq 0$, if $A \in \mathcal{U}_n$ and $B(x) \in \mathcal{U}_n$ for every $x \in A$, then $\Pi(A, B) \in \mathcal{U}_n$.

This is all this section needs. When inductive types arrive, §4 extends the theory.

Two of these laws are design choices. The first is that *application of the point is the point*. In the model a proof of a proposition is the point, and a proof of $\forall x : A. P$ is again the point, not a function. The law makes application one operation that does the right thing on both kinds of value, so the interpretation of

f a need not know which kind it is applying. The second is that *a graph is never the point*: a function and a proof are never the same set. A value that is known to be in a function space, or merely known not to be the point, is therefore a graph, and by domain uniqueness a graph carries its own domain. This is what lets β fire with no premise at all.

One law is deliberately restricted: closure of a universe under function spaces is stated for $\mathcal{U}_1, \mathcal{U}_2, \dots$ and not for $\mathcal{U}_0$. A proposition $\forall x : A. P$ has no function space in the model. It has a truth value, and a truth value is in $\mathcal{U}_0$ whatever A is. This is the model's impredicativity.

2.5. The interpretation

Every term denotes a set. The interpretation is *total*, defined on every term, well-typed or not, and *term-directed*: it is a structural recursion on the term, consulting no derivation and no type. It takes two parameters:

- a valuation φ of the level parameters (Section 2.2);
- a *variable environment* ρ , assigning a set to every variable in scope.

We write $\llbracket e \rrbracket_\rho$ for the set the term e denotes under ρ (φ stays implicit), and $\rho, x \mapsto v$ for ρ extended with the value v for the variable x . The clauses:

$$\begin{aligned}\llbracket x \rrbracket_\rho &= \rho(x) \\ \llbracket \text{Sort } u \rrbracket_\rho &= \mathcal{T}_{\varphi(u)} \\ \llbracket f \ a \rrbracket_\rho &= \llbracket f \rrbracket_\rho \cdot \llbracket a \rrbracket_\rho \\ \llbracket \lambda x : A. b \rrbracket_\rho &= \Lambda(\llbracket A \rrbracket_\rho, v \mapsto \llbracket b \rrbracket_{\rho, x \mapsto v}) \\ \llbracket \forall x : A. B \rrbracket_\rho &= \Pi^v(\llbracket A \rrbracket_\rho, v \mapsto \llbracket B \rrbracket_{\rho, x \mapsto v}) \\ \llbracket \text{let } x : T := v; b \rrbracket_\rho &= \llbracket b \rrbracket_{\rho, x \mapsto \llbracket v \rrbracket_\rho}\end{aligned}$$

with $\varphi(u)$ the value of the level u at the valuation, and $\mathcal{T}_n$ the sets a sort denotes, below. The two binder clauses choose between two readings by the values:

$$\begin{aligned}\Pi^v(A, B) &= \begin{cases} \Pi_0(A, B) & \text{if } B(v) \text{ is a truth value for every } v \in A, \\ \Pi(A, B) & \text{otherwise;} \end{cases} \\ \Lambda(A, F) &= \begin{cases} \text{pt} & \text{if } F(v) = \text{pt} \text{ for every } v \in A, \\ \text{graph}(F, A) & \text{otherwise.} \end{cases}\end{aligned}$$

Here $\Pi_0(A, B)$ is the propositional function space, $\text{tv}(\forall v \in A. B(v) = \{\text{pt}\})$. When the fibres $B(v)$ are truth values it is their intersection $\bigcap_{v \in A} B(v)$ (the empty intersection being $\{\text{pt}\}$), which is $\{\text{pt}\}$ exactly when every fibre is and $\emptyset$ otherwise: the function space of subsets of a singleton. A let reads its body at the value's value, and its type is not read, as no type is read anywhere.

So a binder has two readings. When the body is not a proposition, a lambda is a graph and a $\forall$ is a set of such graphs, a function space. When it is, a lambda is a proof, hence the point, and a $\forall$ is a proposition, true when every *fibre* (the set $\llbracket B \rrbracket_{\rho, x \mapsto v}$ at a $v \in \llbracket A \rrbracket_\rho$) is $\{\text{pt}\}$. Which reading applies is decided by the values the body takes over the domain, and by nothing else. The interpretation does not know the sort of B , and does not compute it.

The empty domain. Over a non-empty domain the values choose the reading the sort of the body would choose. Over an *empty* domain there are no values, and both clauses take their first case: a function space over an empty domain is $\{\text{pt}\}$, and an abstraction over one is the point, whatever the sort of the body. So $\text{False} \rightarrow \text{Nat}$, a type of sort Type , denotes $\{\text{pt}\}$, and its inhabitant $\text{fun } h \Rightarrow h.\text{elim}$ denotes the point. A function space over an inhabited domain can do the same: $\text{Nat} \rightarrow \text{False} \rightarrow \text{Nat}$ has $\{\text{pt}\}$ as every fibre and is itself $\{\text{pt}\}$, and $\text{fun } n \ h \Rightarrow h.\text{elim}$ is the point. No other reading over the empty domain does better.

Whether the codomain is a proposition there is a fact about the syntax, not about any value, and reading an empty Π -type as a function space instead would make the proposition $\text{False} \rightarrow \text{False}$ denote a one-element set of graphs, which no truth value is.

What a sort denotes. A set is a *type* when it is a truth value or *strict*, that is, neither the point nor a set holding the point. The members of a type are all the point or none of it, and this is the one fact about types the soundness proof reads. A sort denotes the types of its universe: $\mathcal{T}_0 = \mathcal{U}_0$, the truth values, and $\mathcal{T}_n$ for $n \geq 1$ the members of $\mathcal{U}_n$ that are types. These are cumulative, and $\mathcal{T}_{n+1}$ holds the truth values too. It must, since a function space over an empty domain is $\{\text{pt}\}$, which is no strict set. The point is no type, so no sort holds it.

IN CON-LECHE Con-leche's sorts denote the same sets: a sort's universe is the universe of types, read off the same notions of a strict set and a type.

This is the whole of the interpretation. Nothing had to be well-typed, no typing judgement was consulted, and there is no partiality to discharge. The next subsection says what replaces typing. Three lemmas about the interpretation are needed later, each proved by induction on the term. Substituting a term for a variable is extending the environment with the term's value, $\llbracket b[x := a] \rrbracket_\rho = \llbracket b \rrbracket_{\rho, x \mapsto \llbracket a \rrbracket_\rho}$; hence a let denotes what its ζ -reduct denotes, at every environment; and a term does not see a variable it does not mention.

2.6. The semantic invariant

There is no typing judgement in the proof. In its place is a predicate on terms, the *semantic invariant*: a term is well-denoted when its applications and binders respect domains. It is stated under a valuation φ and an environment ρ , like the interpretation, by recursion on the term: it holds of a term when it holds of the subterms and one condition on the term's own shape is met. We write $\rho \models e$ for “ e is well-denoted under ρ ” (φ is fixed throughout). Variables and sorts are always well-denoted, and

$$\begin{array}{c}
\frac{\rho \models f \quad \rho \models a \quad \llbracket f \rrbracket_\rho \in P \quad \llbracket a \rrbracket_\rho \in A'}{\rho \models f a} \text{ APP} \qquad \frac{\rho \models v \quad \rho, x \mapsto \llbracket v \rrbracket_\rho \models b}{\rho \models \text{let } x : T := v; b} \text{ LET} \\
\\
\frac{\forall v \in \llbracket A \rrbracket_\rho. \rho, x \mapsto v \models b}{\rho \models \lambda x : A. b} \text{ LAM} \\
\\
\frac{\rho \models A \quad \llbracket A \rrbracket_\rho \text{ a type} \quad \forall v \in \llbracket A \rrbracket_\rho. \llbracket b \rrbracket_{\rho, x \mapsto v} \in B(v) \text{ a type}}{\rho \models \lambda x : A. b} \text{ LAM} \\
\\
\frac{\forall v \in \llbracket A \rrbracket_\rho. \rho, x \mapsto v \models B}{\rho \models \forall x : A. B} \text{ PI} \\
\\
\frac{\rho \models A \quad \llbracket A \rrbracket_\rho \text{ a type} \quad \forall v \in \llbracket A \rrbracket_\rho. \llbracket B \rrbracket_{\rho, x \mapsto v} \text{ a type}}{\rho \models \forall x : A. B} \text{ PI}
\end{array}$$

In app, P is either a function space $\Pi(A', B')$ whose domain A' is a type, or a propositional function space $\Pi_0(A', B')$ whose fibres are truth values. In lam, B is some family of types bounding the body's values, its *codomain*. In words: an application applies a function, or a proof of a propositional $\forall$, to a member of its domain; a binder's domain is a type; a lambda's values lie in a bounded family of types, and a $\forall$'s fibres are types. The invariant records which sets are types, and that is what makes the choice between the two readings agree with membership. An abstraction is a member of the function space of a family of types bounding its values: if every fibre is a truth value, every value is the point and so is the abstraction; if one fibre is not, it is strict, the body's value there is not the point, and the abstraction is the graph.

The semantic invariant is transported by substitution: $b[x := a]$ is well-denoted under ρ exactly when b is well-denoted under $\rho, x \mapsto \llbracket a \rrbracket_\rho$, provided a itself is. The proof is by induction on the term, using the interpretation's substitution lemma; no typing derivation is involved.

A context Γ , the list of the types of the variables in scope, is *satisfied* by ρ when every entry is well-denoted under the environment beyond it, denotes a type there, and has the variable's value as a member. This

is the semantic reading of “ Γ is a well-formed context”, and it is the only thing the claims below assume about the context.

The first non-routine step is β . Take a well-denoted redex $(\lambda x : A. b) a$. From the application rule we know that the lambda’s value lies in some P whose domain A' contains $\llbracket a \rrbracket_\rho$; from the lambda rule we know that the body is bounded over $\llbracket A \rrbracket_\rho$. For the β law we need $\llbracket a \rrbracket_\rho \in \llbracket A \rrbracket_\rho$, the argument in *the lambda’s own* domain, and the semantic invariant has given us A' , not A .

Lemma 1 (β where the redex is not the point). If $(\lambda x : A. b) a$ is well-denoted under ρ and does not denote the point, then $\llbracket (\lambda x : A. b) a \rrbracket_\rho = \llbracket b[x := a] \rrbracket_\rho$ and $b[x := a]$ is well-denoted under ρ .

Proof. The application is not the point, so neither is the lambda, since $\text{pt} \cdot a = \text{pt}$. An abstraction that is not the point is a graph, $\text{graph}(v \mapsto \llbracket b \rrbracket_{\rho, x \mapsto v}, \llbracket A \rrbracket_\rho)$. A graph is never the point, so the application’s P is a function space $\Pi(A', B')$, not a propositional function space. The lambda’s own rule puts the same graph into $\Pi(\llbracket A \rrbracket_\rho, B)$ by introduction, and domain uniqueness gives $A' = \llbracket A \rrbracket_\rho$; so $\llbracket a \rrbracket_\rho \in \llbracket A \rrbracket_\rho$, and the β law computes the application to $\llbracket b \rrbracket_{\rho, x \mapsto \llbracket a \rrbracket_\rho}$, which is $\llbracket b[x := a] \rrbracket_\rho$ by the substitution lemma. The semantic invariant of the reduct is the substitution transport. $\square$

For a redex that does denote the point there is no such lemma. Its lambda may be the point, which carries no domain, and its argument may lie anywhere. Such a redex is well-denoted, and its reduct need not be: $(\lambda x : A. g(k x)) a$, with a lambda that is the point and an argument a outside the domain A of the graph k , reduces to $g(k a)$, which is not well-denoted. So the reduction claim below says nothing about a term that denotes the point, and β needs no premise. Section 2.7 shows that the other rules need no more than such a claim.

Why no sort holds the point. The claims below rely on one fact: an inferred type is a type, so it is never the point. It also keeps β sound on stored values. A stored constant $c : \text{Sort } 1$ whose value is the ill-typed redex $(\lambda h : (\forall P : \text{Prop}. P). h) \text{Prop}$ reduces by δ and β to Prop . The redex denotes the point (its lambda is an abstraction over an empty domain), and Prop does not. Such an environment has no model: the constant’s set would have to be the point and a member of $\mathcal{T}_1$, which holds no point.

2.7. The three claims and their proof

Theorem 2 (Soundness of the three relations). Fix a valuation φ and let ρ satisfy Γ . Then:

1. reduction preserves the denotation and the semantic invariant of a term that is not the point: if $\Gamma \vdash e \rightsquigarrow e'$, e is well-denoted and $\llbracket e \rrbracket_\rho \neq \text{pt}$, then e' is well-denoted and $\llbracket e \rrbracket_\rho = \llbracket e' \rrbracket_\rho$;
2. equality means equal sets where neither side is the point: if $\Gamma \vdash a \equiv b$, both a and b are well-denoted and neither denotes the point, then $\llbracket a \rrbracket_\rho = \llbracket b \rrbracket_\rho$;
3. inference establishes the semantic invariant and a membership: if $\Gamma \vdash e \Rightarrow T$, then e and T are well-denoted, $\llbracket T \rrbracket_\rho$ is a type, and $\llbracket e \rrbracket_\rho \in \llbracket T \rrbracket_\rho$.

(The theorem is soundness, with the three claims stated.)

Reduction and equality assume the semantic invariant; inference establishes it. The inference rules are the ones that check a term’s shape, so their soundness proves that the term’s applications and binders respect domains; the other two relations are handed well-denoted terms and pass the semantic invariant along. The first two claims are also conditional on the point: they say nothing about a term that denotes it.

Why the claims may say nothing about the point. A claim that is silent about the point is only useful if every rule that uses a premise’s claim has that premise not the point, or does not need the claim there. Every rule falls into one of four cases.

1. *The subject is a type, never the point.* The reductions of an inferred type to a sort or a $\forall$ (in the binder rules, in app, in η , in proof irrelevance), the comparisons of two types (an argument’s type with a domain in app, a domain with a domain in the congruences, the checker’s type comparisons in ι), the

fibres compared in $\forall$ -congruence. The third claim made each subject a type, so the conditional claim applies outright.

2. *The subject is not the point by hypothesis.* trans passes the first claim's denotation on to the second; head reduces the head of an application that is not the point, which is therefore not the point either; red-l reduces a side of an equality that is not the point; app compares the heads of two applications that are not the point.
3. *The two sides share a type, so they are both the point or neither.* app's arguments: the heads are one graph, whose domain is a type and holds both arguments. ι 's comparisons and its major premise (Section 4) are of the same kind. The bodies of lambda congruence and η are the one place where nothing else supplies a common type, and that is why those rules compare the bodies' types (Section 2.8).
4. *The step is an identity everywhere.* β on a redex that is not the point (Lemma 1), ζ and δ (Section 3) claim nothing beyond what holds at every environment.

Corollary 3. If the checker infers $\vdash e \Rightarrow T$ in the empty context, then $\llbracket e \rrbracket_\rho \in \llbracket T \rrbracket_\rho$ at every valuation and every ρ .

This is the statement the environment section builds on: a declaration is accepted when its value's inferred type is definitionally equal to its declared type, and the corollary, with the second claim, puts the value's set into the declared type's set.

Proofs that are not proofs. The proof asks whether a term's value is the point, never whether the term is a proof in Lean's sense (except in the proof-irrelevance rule, whose claim turns out to be empty), and it learns that only through a type the term lies in. So a term whose value is the point is treated like a proof whatever the sort of its type, also when its value is the point only at some values of a variable.

The three claims are proved together, by one structural induction over the three mutually inductive relations (as in con-leche). Every rule is one case, and every case is a lemma about that rule alone, with the induction hypothesis for each premise as an assumption. The routine cases are listed at the end; the others follow.

Proof. **Rule beta**, $(\lambda x : A. b) a \rightsquigarrow b[x := a]$:

This is Lemma 1. The rule has no premise, and the lemma needs none, because the claim is asked only where the redex is not the point.

Rule zeta, let $x : T := v; b \rightsquigarrow b[x := v]$:

The two sides denote the same set at every environment, the point included, and the reduct's semantic invariant is the let 's, through the substitution transport.

Rule red-l, $\Gamma \vdash a \equiv b$ from $\Gamma \vdash a \rightsquigarrow a'$ and $\Gamma \vdash a' \equiv b$:

a is not the point, so by the first claim a' is well-denoted and $\llbracket a \rrbracket_\rho = \llbracket a' \rrbracket_\rho$. Now a' and b are both well-denoted and neither is the point, so the second claim applies to the continuation, and the two equalities chain. This is the one sound way to chain in the equality relation, because a reduction step *produces* the semantic invariant of its result; see the discussion of transitivity below.

Rule sort, $\text{Sort } u \equiv \text{Sort } v$ when $u \doteq v$:

The oracle is assumed correct: it answers yes only if the levels agree at every valuation (Section 2.2), so the two sorts denote the same set.

Rule pi, $\forall x : A_1. B_1 \equiv \forall x : A_2. B_2$ from $\Gamma \vdash A_1 \equiv A_2$ and $\Gamma, x : A_2 \vdash B_1 \equiv B_2$:

Both $\forall$ s are well-denoted, so their domains are types, never the point, and the second claim on the domains gives $\llbracket A_1 \rrbracket_\rho = \llbracket A_2 \rrbracket_\rho$. At every v in that domain the fibres are types too, so the claim on the bodies makes them equal. A $\forall$'s set is a function of its domain's set and its fibres, the choice of reading included, since that is read off the fibres; so the two $\forall$ s are equal. No comparison of the two sides' sorts is needed.

Rule lam, $\lambda x : A_1. b_1 \equiv_\lambda \lambda x : A_2. b_2$ from the domains, the bodies, and the bodies' types $t_1 \equiv t_2$:

The domains are equal as for $\forall$. Fix v in the domain. The third claim puts $\llbracket b_1 \rrbracket$ in $\llbracket t_1 \rrbracket$ and $\llbracket b_2 \rrbracket$ in $\llbracket t_2 \rrbracket$

(at $\rho, x \mapsto v$), both types, and the second claim on the types, which are not the point, makes them one type. Its members are all the point or none, so either both bodies are the point, or neither is and the claim on the bodies makes them equal. Either way they are equal at every v , and so are the two abstractions, whether or not either of them is the point. The claim is therefore unconditional; lam-outer passes such a claim outwards (Section 2.8), and lam weakens it to the second claim. Without the type premises the argument breaks at “both or neither”: a body may be the point at some v and not at others, and the claim on the bodies says nothing at such a v .

Rule app, $f_1 a_1 \equiv f_2 a_2$:

Neither application is the point, so neither head is, and the claim on the heads makes them one graph. Each application’s rule puts its head in some P , with its argument in P ’s domain. A graph is in no propositional function space, so each P is a function space, and domain uniqueness puts both arguments in one domain, a type. They are both the point, or neither is and the claim on the arguments makes them equal.

Rule fun-eta, $\lambda x : A_1. b_1 \equiv b$ when $\Gamma \vdash b \Rightarrow T \rightsquigarrow \forall x : A_2. B, \Gamma \vdash A_2 \equiv A_1, \Gamma, x : A_1 \vdash b_1 \equiv b x$, and the body’s type agrees with B :

b is not the point, so the third claim, then the first, put $\llbracket b \rrbracket_\rho$ in the denotation of the $\forall$, which is well-denoted; the domains are equal as before. Fix v in the domain. The lambda’s body lies in its inferred type, which the type premise makes the fibre $\llbracket B \rrbracket_{\rho, x \mapsto v}$; the term $b x$ lies there by elimination. In one type, they are both the point or neither, and in the second case the claim on the bodies makes them equal. So the lambda denotes the abstraction of $v \mapsto \llbracket b \rrbracket_\rho \cdot v$, and a member of a function space of types is the abstraction of its applications.

Rule proof-irrel, $a \equiv b$ when $\Gamma \vdash a \Rightarrow T_a \Rightarrow S_a \rightsquigarrow \text{Sort } u$ with $u \doteq 0$, and likewise for b :

By the third claim twice and the first once, $\llbracket a \rrbracket_\rho \in \llbracket T_a \rrbracket_\rho \in \mathcal{T}_0$, a truth value, whose only possible member is the point. So a denotes the point, and the claim, which is about sides that are not the point, holds vacuously. The two types T_a and T_b were never compared, and the semantic invariants of a and b were not used.

Rule pi, inference, $\Gamma \vdash \forall x : A. B \Rightarrow \text{Sort } (\text{imax}(u, v))$ when $\Gamma \vdash A \Rightarrow S \rightsquigarrow \text{Sort } u$ and $\Gamma, x : A \vdash B \Rightarrow T \rightsquigarrow \text{Sort } v$:

The third claim for A and the first for S put $\llbracket A \rrbracket_\rho$ in $\mathcal{T}_{\varphi(u)}$, so it is a type, and for every $v' \in \llbracket A \rrbracket_\rho$ the environment $\rho, x \mapsto v'$ satisfies $\Gamma, x : A$; the same claims for B put every fibre in $\mathcal{T}_{\varphi(v)}$. That is the $\forall$ rule of the semantic invariant, and $\Pi^v(A, B)$ lies in $\mathcal{T}_{\varphi(\text{imax}(u, v))}$. If $\varphi(v) = 0$, every fibre is a truth value, so is $\Pi^v(A, B)$, and $\varphi(\text{imax}(u, v)) = 0$. Otherwise $\Pi^v(A, B)$ is a truth value or a function space in $\mathcal{U}_{\text{max}(\varphi(u), \varphi(v))}$ by cumulativity and closure, a type either way. $\Pi^v(A, B)$ need not be a function space when $\varphi(v) \neq 0$: over an empty domain it is $\{\text{pt}\}$, and $\mathcal{T}$ admits it.

Rule lam, inference, $\Gamma \vdash \lambda x : A. b \Rightarrow \forall x : A. B$ when $\Gamma \vdash A \Rightarrow S \rightsquigarrow \text{Sort } u$ and $\Gamma, x : A \vdash b \Rightarrow B$:

The same argument one level down. The domain is a type, and for every v' in it the third claim for b gives b and B well-denoted, $\llbracket B \rrbracket$ a type and $\llbracket b \rrbracket \in \llbracket B \rrbracket$. So the family of the $\llbracket B \rrbracket$ is a bounded codomain of types, which is the lambda rule, and the $\forall$ rule of the inferred type is met too; and the introduction law for values puts the abstraction into $\Pi^v(A, B)$. Nothing about the sort of B is needed, so the rule computes none.

Rule app, inference, $\Gamma \vdash f a \Rightarrow B[x := a]$ when $\Gamma \vdash f \Rightarrow T \rightsquigarrow \forall x : A. B, \Gamma \vdash a \Rightarrow T_a$ and $\Gamma \vdash T_a \equiv A$:

By the third claim, f and T are well-denoted and $\llbracket f \rrbracket_\rho \in \llbracket T \rrbracket_\rho$; T is a type, so by the first claim the $\forall$ is well-denoted and $\llbracket T \rrbracket_\rho = \llbracket \forall x : A. B \rrbracket_\rho$. By the third claim for a , a and T_a are well-denoted, $\llbracket T_a \rrbracket$ is a type and $\llbracket a \rrbracket_\rho \in \llbracket T_a \rrbracket_\rho$; A is a type by the $\forall$ rule, so the second claim gives $\llbracket T_a \rrbracket_\rho = \llbracket A \rrbracket_\rho$ and $\llbracket a \rrbracket_\rho \in \llbracket A \rrbracket_\rho$. Now $f a$ is well-denoted by the application rule of Section 2.6, with the $\forall$ ’s denotation as $\Pi^v(A, B)$; $B[x := a]$ is well-denoted by the substitution transport; and the elimination law puts $\llbracket f \rrbracket_\rho \cdot \llbracket a \rrbracket_\rho$ into the fibre at $\llbracket a \rrbracket_\rho$, which is $\llbracket B[x := a] \rrbracket_\rho$ by the substitution lemma.

A syntactic proof would here invert a derivation of $f : T$ to learn the domain and the codomain, using the injectivity of $\forall$ that Section 2.3 discussed. Our proof inverts nothing. The checker itself reduced T to a syntactic $\forall$, the first claim says that the reduction did not change the set, and that set is $\Pi^v(A, B)$ by the interpretation’s rule.

Rule let, inference. $\Gamma \vdash \text{let } x : T := v; b \Rightarrow B$ when $\Gamma \vdash b[x := v] \Rightarrow B$ and the checks on the value: The claim for the substituted body is the claim for the let : the two denote the same set, and the let ’s semantic invariant is the substituted body’s, by the transport. The two checks on the value are not used. They would be if the body were inferred at a variable of type T instead: then the value must lie in the type to extend a satisfying environment, and the checks give that.

The rest, by induction on the derivation.

- Reduction: the no-step reduction is $\llbracket e \rrbracket_\rho = \llbracket e \rrbracket_\rho$; trans chains two reductions, passing the semantic invariant along; head reduces the head of an application that is not the point, which is not the point either, and keeps the application’s rule, because the head’s set did not change.
- Equality: refl is again $\llbracket e \rrbracket_\rho = \llbracket e \rrbracket_\rho$, and sym swaps the two sides.
- Inference: a variable’s type is read off the satisfied context; a sort’s type is the next sort, which contains it.

□

Why there is no transitivity. The proof above shows why the rule “ $a \equiv b$ and $b \equiv c$ give $a \equiv c$ ” cannot be added. The second claim assumes both sides well-denoted. In a transitivity case the induction would have to apply the hypothesis to $a \equiv b$, and for that it needs b well-denoted. But b is neither a subterm of a or c nor produced by a premise, so nothing supplies its semantic invariant. Every other rule keeps one discipline: each term of an equality premise is either a subterm of the conclusion (the congruences, the η body) or a term that another premise produced, a reduct (red-l) or an inferred type (lam, η), and the corresponding claim delivers the semantic invariant of every produced term. The one way to chain is therefore “reduce, then continue”, and that is how the checker’s equality test is structured: it head-normalises a side and compares again.

2.8. The certification tax

The checker makes two kinds of check that other checkers for Lean (the official kernel, Lean4Lean, nanoda) skip. Both come from one fact: the semantic invariant admits terms that are well-denoted but not well-typed. A typing judgement would exclude them, and with them the need for the checks. The semantic proof has no typing judgement, so the checker verifies the one property the proof would otherwise take from typing. The two kinds are:

1. **The bodies’ types at lambda congruence and η :** inferred by full inference and compared, at the innermost level of two lambda telescopes only. This subsection explains them.
2. **The constructor application in one case of ι -reduction:** when the family of the major premise may be a proposition at the instance and the rule is not K-like (one constructor, no fields, in Prop; Section 4.2.8), the checker compares the constructor’s levels, parameters and index expressions with the recursor’s, and type-checks every parameter and field of the constructor application against the constructor’s type. The official kernel makes none of these checks at an ι step; Section 4.2.7 and Section 4.2.8 explain them.

IN CON-LECHE The checks are the checker’s innermost lambda level, its η and the case of ι . On real input neither kind ever fails: they never change a verdict on a stream the official kernel accepts. Both run at --verified only; --trusted skips them.

Why lambda congruence needs the bodies’ types. The claim on the bodies says nothing where a body is the point, and whether a body is the point can vary with the variable. Without a common type of the two bodies the rule is unsound in the model. Take the two lambdas

$$\lambda(x : \text{Prop}). (\lambda(h : E\ x). h)\ n \quad \text{and} \quad \lambda(x : \text{Prop}). n,$$

with E sending `True` to `Prop` and `False` to `False`, and n denoting `True`. They have equal domains, and their bodies are equal by β then refl ; both premises are derivations of the checker. Yet at $x := \text{False}$ the left body is an abstraction over an empty domain, applied, hence the point, while the right body is `True`. The two lambdas are well-denoted, neither is the point, and they differ. The left body is ill-typed (n is no member of $E\ x$), but it is well-denoted, because the abstraction over $E\ \text{False}$ is the point and constrains nothing.

Why other checkers do without it. Their argument only ever compares two terms of one type: the caller compared the types, or knows them, and lambda inversion gives the two bodies a common type, by subject reduction and uniqueness of types. The model has neither, and the semantic invariant is satisfied by terms no typing judgement derives, so the common type is recomputed.

Innermost only. The checker compares two lambda telescopes level by level, and infers and compares the bodies' types only at the innermost level, where the descent through two syntactic lambdas stops (the next comparison is η , or a reduction, or any other rule). At the outer levels the domains are compared and nothing else. This is sound because the type comparison makes the claim at the innermost level *unconditional*: two bodies in one type are equal at every value of the variable, the point included. An unconditional equality of the bodies at every value is an equality of the abstractions, with no type at all, so the claim propagates outwards through any number of levels. A δ step or a reduction between two levels ends the telescope, and the level below it is an innermost level again.

Full inference. The bodies' types are inferred by full inference (Section 2.3.3), not by `infer_only` (Section 2.9). With `infer_only` there is a typed definition at which every premise of lambda congruence holds while the two lambdas differ. Every premise is a derivation of the relations, so the rule itself is unsound, whatever claim `infer_only` makes.

2.9. The cheaper mode of inference

The official kernel infers the type of a term it has already checked by a cheaper mode, `infer_only`. The fragment has that mode: the rules of Section 2.3.3 without `app`'s comparison of the argument's type with the domain, without the lambda's check of its domain, and without the check of a let-binding's value against its type.

Its result is not always a type of the term: $(\lambda h : \text{False}. (h.\text{elim} : \text{Nat}))\ 0$ gets the type `Nat`, is well-denoted, and denotes the point (the lambda is an abstraction over an empty domain), which is no member of the set `Nat` denotes. What holds is a hereditary claim: the value is the point, or a member of the type, or, when the type is a $\forall$, a graph over its domain whose applications satisfy the claim at the $\forall$'s body. It is proved for every rule of `infer_only`.

The claim serves two consumers. One is proof irrelevance with both types inferred by `infer_only` (the rule's case): a value whose type has sort `Prop` is the point, the hereditary case included, because the body of a $\forall$ of sort `Prop` has sort `Prop`. The other is η against a term whose head is a variable or a constant, whose claim is never the hereditary one (the rule's case). The bodies' types at lambda congruence and η are not among them, as Section 2.8 showed, so the checker infers those by full inference and keeps the results of the two modes apart.

IN CON-LECHE The checker's cheaper mode is `infer_only` exactly. Like it, the mode does not check a let-binding's value against its type (full inference does, Section 2.3.3); the soundness proof needs no such check, since the value is substituted.

2.10. The argument in brief

Three points of this section run through the rest of the paper. The claims of reduction and equality are conditional: they say nothing about a term that denotes the point. The checks of the certification tax are exactly what a typing judgement would otherwise supply: a common type for the two sides of a

comparison where nothing else gives one. And the proof uses no subject reduction, no confluence, no injectivity of Π -types and no unique typing; it needs only the laws of the set theory and the derivations of the checker's three relations.

3. Adding definitions

§2 had no environment. This section adds one, together with the constants that refer to its entries, the reduction rule δ that unfolds a definition, and the *model* of the environment, from which the interpretation reads a constant. The environment grows one declaration at a time, and each step is checked by the relations of §2, extended by the rules below. We say what the checker checks at each step, what it stores, and how the model grows with the environment so that its laws keep holding. This section covers definitions; inductive types, the rule ι and the consistency corollary follow in Section 4.

The environment E , fixed throughout like the valuation, is a list of the constants accepted so far, most recent first. A stored constant has its level parameters $\vec{p}$, its type (a closed term over $\vec{p}$) and its *kind*. We write $(c.\{\vec{p}\} : T) \in E$ for “ c is stored with parameters $\vec{p}$ and type T ”, and $(c.\{\vec{p}\} : T := v) \in E$ when it is a definition with value v . A *definition* carries a value. The other kinds (type former, constructor, recursor) are what an inductive block stores, and Section 4.1 introduces them. A name is stored at most once.

Constants. The grammar of Section 2.1 gains one form: a constant c of the environment, used at a list of levels $\vec{\ell}$, one per level parameter of its declaration.

$$e ::= \dots \mid c.\{\vec{\ell}\}$$

Level instantiation $e[\vec{p} := \vec{\ell}]$, used when a constant declared with level parameters $\vec{p}$ is taken at the levels $\vec{\ell}$, replaces the parameters in sorts and in the level lists of constants (as in con-leche).

The rules for constants. A constant has its declared type at the levels it is used at, and two constants of the same name are compared through the level oracle (in con-leche: inference and equality).

$$\frac{(c.\{\vec{p}\} : T) \in E \quad |\vec{\ell}| = |\vec{p}|}{\Gamma \vdash c.\{\vec{\ell}\} \Rightarrow T[\vec{p} := \vec{\ell}]} \text{CONST} \quad \frac{\vec{\ell} \doteq \vec{\ell}'}{\Gamma \vdash c.\{\vec{\ell}\} \equiv c.\{\vec{\ell}'\}} \text{CONST}$$

The δ rule. A definition unfolds to its value at the levels it is used at (as in con-leche). With the head rule of Section 2.3, an applied definition unfolds at its head.

$$\frac{(c.\{\vec{p}\} : T := v) \in E \quad |\vec{\ell}| = |\vec{p}|}{\Gamma \vdash c.\{\vec{\ell}\} \rightsquigarrow v[\vec{p} := \vec{\ell}]} \text{DELTA}$$

What a definition must satisfy. Before the checker stores a definition c with parameters $\vec{p}$, type T and value v , it checks four things (in con-leche: the common checks and the value check):

- the name c is fresh;
- the type has a sort: $\vdash T \Rightarrow S \rightsquigarrow \text{Sort } u$;
- the value's inferred type is definitionally equal to the declared type: $\vdash v \Rightarrow T'$ and $\vdash T' \equiv T$;
- both terms use only the level parameters $\vec{p}$.

A stored term is thus closed, mentions only stored constants and uses only the declared level parameters. These three facts let the model read a stored term without looking at anything that is added later, because the interpretation of a term depends only on the constants and the free variables it mentions and the level parameters it uses (constants, variables, parameters).

The interpretation, extended. The interpretation of Section 2.5 gains one parameter, an *assignment* M of a set to every constant at every list of concrete levels, and one clause: a constant denotes what the assignment says,

$$\llbracket c.\{\vec{\ell}\} \rrbracket_\rho = M(c, \varphi(\vec{\ell}))$$

with $\varphi(\vec{\ell})$ the list of the levels' values. It stays term-directed. M is a parameter like ρ , consulted at a constant the way ρ is at a variable. The environment E itself, with its stored types and values, is never read, and no derivation and no type is needed. A fourth lemma joins those of Section 2.5, by the same induction: instantiating level parameters is changing the valuation. And the semantic invariant gains the clause that a constant, like a variable or a sort, is always well-denoted.

The contract. What the soundness theorem, extended below, assumes of the environment is a model: an assignment $M(c, \vec{n})$ of a set to every constant c and every list of natural numbers $\vec{n}$ (the values of its level parameters), satisfying two laws (four more, for inductive types, join in Section 4). Throughout, $\vec{\ell}$ ranges over level lists with $|\vec{\ell}| = |\vec{p}|$, φ over valuations and ρ over variable environments, and $\rho \models e$ is the semantic invariant of Section 2.6.

Definition 4 (Model of an environment).

1. Typing: for every $(c.\{\vec{p}\} : T) \in E$,

$$\rho \models T[\vec{p} := \vec{\ell}], \quad \llbracket T[\vec{p} := \vec{\ell}] \rrbracket_\rho \text{ is a type, and } M(c, \varphi(\vec{\ell})) \in \llbracket T[\vec{p} := \vec{\ell}] \rrbracket_\rho.$$

2. Unfolding: for every $(c.\{\vec{p}\} : T := v) \in E$,

$$\rho \models v[\vec{p} := \vec{\ell}] \quad \text{and} \quad \llbracket v[\vec{p} := \vec{\ell}] \rrbracket_\rho = M(c, \varphi(\vec{\ell})).$$

In words: every stored constant is a member of its type, which is a type in the sense of Section 2.5, and a definition's value denotes the constant, at every instantiation of the declared level parameters.

| IN CON-LECHE The real proof's model carries the same two laws among others (the carrier's invariant).

The empty environment has a model trivially: any assignment, and laws with nothing to say.

Soundness, extended. [Theorem 2](#) holds for the relations extended by the three rules above, with one more hypothesis: fix a model of the environment, a valuation φ , and let ρ satisfy Γ ; then the three claims hold as stated, and [Corollary 3](#) holds under every model. The induction of Section 2.7 gains three cases, one per rule.

Proof. **Rule delta**, $c.\{\vec{\ell}\} \rightsquigarrow v[\vec{p} := \vec{\ell}]$ for a definition c with parameters $\vec{p}$ and value v , at $|\vec{\ell}| = |\vec{p}|$ levels: Law 2 says the instantiated value is well-denoted and denotes $M(c, \varphi(\vec{\ell}))$, which is what the constant denotes. The redex's semantic invariant is not used. [Theorem 5](#) shows the law holds when a definition is added.

Rule const, equality, $c.\{\vec{\ell}\} \equiv c.\{\vec{\ell}'\}$ when $\vec{\ell} \doteq \vec{\ell}'$ pointwise:

The oracle answers yes only if the levels agree at every valuation (Section 2.2), so the two constants read the same entry of M .

Rule const, inference, $c.\{\vec{\ell}\} \Rightarrow T[\vec{p} := \vec{\ell}]$:

The constant is well-denoted, and law 1 says its instantiated type is well-denoted, a type, and contains the constant's set. $\square$

| **Theorem 5 (Installing a definition).** If the environment has a model and the definition c passes the checks above, then the environment extended with c has a model (as in con-leche).

Proof. Let M be the model. Define M' as M on every old constant and, at c and a list $\vec{n}$, as the set $\llbracket v \rrbracket$ read at the valuation that sends $\vec{p}$ to $\vec{n}$. Since v is closed and uses no other parameter, no ρ and no other part of φ enters. The old laws hold for M' because no stored term mentions c . For the new constant we use [Corollary 3](#) and the second claim of [Theorem 2](#). From $\vdash v \Rightarrow T'$ the value and T' are well-denoted and $\llbracket v \rrbracket \in \llbracket T' \rrbracket$, a type; from $\vdash T \Rightarrow S \rightsquigarrow \text{Sort } u$ the type T is well-denoted and a type; two types are never

the point, so the second claim applies to $\vdash T' \equiv T$ and gives $\llbracket v \rrbracket \in \llbracket T \rrbracket$. This holds at every valuation, and instantiating the level parameters is the same as changing the valuation (above), so it holds at every $\vec{\ell}$. Finally, v and T are closed and mention no constant but old ones, because the derivations are in the empty context, where no rule types a variable and the constant rule requires the constant stored. So the two sets are the same under M' as under M . That is law 1, and law 2 is the definition of M' at c . $\square$

The proof is short because §2 did the work. The checks a definition passes are the premises of the corollary, and the corollary's conclusion is law 1.

4. Adding inductive types

The second kind of declaration is an inductive block: a type former with its constructors and its recursor. It is checked and installed by the same shape of argument as a definition (Section 3), and its reduction rule ι reads the environment as δ does. Two pieces of work are new. The constructed sets (the family, the constructors, the recursor) must be members of their generated types, which is law 1. And the ι law must be stated and proved: it is a third law of the contract of Section 3, and three more laws let most ι steps go without checking the constructor application. This section says what the checker checks for a block and what it stores, how the model grows by a least fixed point so that the laws keep holding, when ι needs a certificate, and then the consistency corollary.

4.1. What is checked

An inductive block declares an *indexed family* by its constructors: one type, or one proposition, per choice of indices. At a valuation the family is *type-valued* or *proposition-valued*, as its result sort evaluates to a positive number or to 0. These are the two interpretations of a binder from §2, decided for a whole family at once. The fragment takes a block with one type former, neither mutual nor nested; nested blocks are the subject of Section 5. The block is given by a *specification* (as in con-leche):

- a name I and level parameters $\vec{p}$;
- a *parameter* telescope $(x_1 : P_1) \dots (x_k : P_k)$ and, under it, an *index* telescope $(y_1 : J_1) \dots (y_m : J_m)$; the parameters are the same for every constructor and every field, the indices vary;
- a result sort $\text{Sort } u$: the family has type

$$\forall \vec{x} : \vec{P}. \forall \vec{y} : \vec{J}. \text{Sort } u;$$

- constructors $c_1, \dots, c_n$, each with a list of *fields* and, under the parameters and the fields, the index expressions $\vec{e}_j$ of its result $I \vec{x} \vec{e}_j$.

A field is one of two kinds (as in con-leche):

- *ordinary*: its domain does not mention I ;
- *reflexive*: its domain is

$$\forall \vec{z} : \vec{A}. I \vec{x} \vec{e},$$

a function, under a telescope of binders $\vec{z}$ whose domains do not mention I , into the family being defined at the block's own parameters and some index expressions. The telescope may be empty; then the domain is $I \vec{x} \vec{e}$, a member of the family itself, and the field is simply *recursive*.

Two conditions of shape come with the kinds. No field's domain mentions I anywhere else; this is strict positivity. And nothing after a reflexive field may depend on its value (as in con-leche and the official kernel), so that the model can read a constructor's domains without knowing the values of its reflexive fields.

IN CON-LECHE Simple, mutual and nested blocks are checked uniformly (§5 of OVERVIEW.md); this section describes that check on a block of the fragment's shape. Its positivity check classifies each field's domain after weak head

normal form, with I at the block's parameters replaced by a variable standing for the family being defined; Section 5 uses the same check (positivity).

Reflexive fields with a non-empty telescope matter to the model. A tree type with a constructor node : $(\text{Nat} \rightarrow \text{Tree}) \rightarrow \text{Tree}$ has nodes with countably many children, so a node may have children of every finite depth, and the set of all trees is larger than the union of the sets of trees of depth k . Iterating the constructors ω times, which reaches every member of Nat , does not reach every tree, and a model must obtain the least fixed point some other way (Section 4.2).

4.1.1. The generated declarations

From the specification the checker generates the types of the type former, the constructors and the recursor, and the recursor's rules. The type former's type is the one displayed above; a constructor's type is

$$\forall \vec{x} : \vec{P}. \forall \vec{f} : \vec{F}. I \vec{x} \vec{e}_j$$

(the two generators). The recursor eliminates into $\text{Sort } \ell$, where the *elimination level* ℓ is either a fresh level parameter (*large elimination*) or 0 (*small elimination*); the elimination rule below says which.

IN CON-LECHE The input is a stream of raw declarations: the type former, the constructors and the recursor as the exporting toolchain stated them. The checker recovers the specification from the type former's and the constructors' types, and checks and stores those two as the stream declares them (a constructor). The recursor it *generates*, as the official kernel does. From the stream's recursor record it takes only the name, the level parameters and the type, which must be definitionally equal to the generated one, and the record's two layout counts, which must equal the generated ones (the comparison). An unverified pre-pass reads off that type which type the recursor eliminates and how its motive and minor premises are laid out (the arguments named on Nat in Example 6); a wrong reading can only make the comparison fail (the pre-pass). The record's rules are never read; the generated recursor and its rules are what is stored (the recursor stage).

Here is the recursor of the natural numbers.

Example 6 (Nat). The block Nat has no parameters, no indices, sort $\text{Sort } 1$, and two constructors: zero with no field and succ with one recursive field (reflexive with the empty telescope). The generated constructor types are $\text{zero} : \text{Nat}$ and $\text{succ} : \forall (n : \text{Nat}). \text{Nat}$. The recursor eliminates into $\text{Sort } \ell$ for a fresh level parameter ℓ (large elimination, see below), and its generated type is

$$\begin{aligned} \text{Nat.rec.}\{\ell\} &: \forall (C : \forall (t : \text{Nat}). \text{Sort } \ell). \\ &\quad \forall (z : C \text{ zero}). \\ &\quad \forall (s : \forall (n : \text{Nat}). \forall (h : C n). C (\text{succ } n)). \\ &\quad \forall (t : \text{Nat}). C t \end{aligned}$$

C is the *motive*; z and s are the *minor premises*, one per constructor, each taking the constructor's fields and, for every recursive field, an *inductive hypothesis* h , the motive at that field; t is the *major premise*. The recursor has two rules, one per constructor: the major premise built by the constructor reduces to the minor premise applied to the fields and to the recursor's own value at each recursive field,

$$\begin{aligned} \text{Nat.rec.}\{\ell\} C z s \text{ zero} &\rightsquigarrow z \\ \text{Nat.rec.}\{\ell\} C z s (\text{succ } n) &\rightsquigarrow s n (\text{Nat.rec.}\{\ell\} C z s n). \end{aligned}$$

What the checker stores as the second rule's right-hand side is the closed term (the binders' domains omitted)

$$\lambda C. \lambda z. \lambda s. \lambda n. s n (\text{Nat.rec.}\{\ell\} C z s n);$$

the ι rule applies it to the recursor's arguments and the constructor's fields, and β does the rest.

The general shape is the same with parameters, indices and telescopes added. The motive takes the indices and a member of the family:

$$C : \forall \vec{y} : \vec{J}. \forall (t : I \vec{x} \vec{y}). \text{Sort } \ell.$$

A minor premise for c_j takes the fields, then one inductive hypothesis per reflexive field. At a field

$$f : \forall \vec{z} : \vec{A}. I \vec{x} \vec{e}$$

the hypothesis is

$$\forall \vec{z} : \vec{A}. C \vec{e} (f \vec{z}),$$

which at the empty telescope is the C n of [Example 6](#). The minor premise ends in

$$C \vec{e}_j (c_j \vec{x} \vec{f}).$$

The recursor takes the parameters, the motive, the minors, the indices and the major, and ends in $C \vec{y} t$. (Generated: the motive, an inductive hypothesis, a minor premise, the recursor's type and a rule's right-hand side; in con-leche: the type, a rule.)

4.1.2. The elimination rule

For a type-valued family, in Sort u with u never zero, the recursor eliminates into any Sort ℓ , ℓ a fresh level parameter: large elimination. For a proposition-valued family the rule is stricter. In the model every proof is the point, so a recursor applied to a proof cannot see which constructor built it or with which fields, and a recursor with a type-valued motive would have to return one value for all of them. The official kernel allows large elimination out of a proposition only under the *subsingleton criterion*: the block has at most one constructor, and every field of that constructor is either itself a proposition or occurs among the constructor's result indices. Then a proof of $I \vec{x} \vec{e}$ determines every field: the propositional ones are all the same proof, and the others can be read off the indices. Otherwise the motive lands in Prop, $\ell = 0$ (*small elimination*). In the fragment the criterion is a condition per field plus the constructor count, required of a large eliminator on a family whose sort *may* be zero (never zero: $1 \leq u$ at every valuation).

IN CON-LECHE The checker asks the constructor count of every family whose sort may be zero (the guard, applied) but the per-field condition only of a family whose sort is provably zero, which is sound because the universe bound (the third check below) makes every field a proposition at any valuation that sends the result sort to zero.

Three of the checks listed below ask whether a sort is Prop, and the checker answers each with the level oracle:

- a field's sort v is a proposition: $v \doteq 0$ (the subsingleton criterion);
- the result sort u is never a proposition: $1 \leq u$ (large elimination);
- a field's sort does not exceed the result sort (the *universe bound*): $v \leq u$, unless $u \doteq 0$, in which case there is no bound (Lean's impredicativity of Prop).

Example 7 (an indexed proposition with large elimination). Let

$$P : \forall (n : \text{Nat}). \text{Prop}$$

have one constructor

$$\text{mk} : \forall (n : \text{Nat}). P \ n.$$

It is a proposition-valued family, indexed by a number, with one ordinary field n that is not a proposition and occurs in the result's index. The subsingleton criterion holds, so the recursor may eliminate into Sort ℓ :

$$\begin{aligned}
& \text{P.rec.}\{\ell\} : \forall(C : \forall(n : \text{Nat}). \forall(t : P\ n). \text{Sort } \ell). \\
& \quad \forall(h : \forall(n : \text{Nat}). C\ n\ (\text{mk } n)). \\
& \quad \forall(n : \text{Nat}). \forall(t : P\ n). C\ n\ t
\end{aligned}$$

with one rule,

$$\text{P.rec.}\{\ell\} C\ h\ n\ (\text{mk } n') \rightsquigarrow h\ n'.$$

The index occurs twice: n is the recursor's index argument, n' the constructor's field. Section 4.2.7 uses this example.

4.1.3. When a block is accepted

A block is accepted when (as in con-leche):

- its names are distinct and fresh;
- the specification is in scope, positivity included;
- the generated type former's type has a type in the current environment;
- each generated constructor's type has a sort in the environment holding the type former, and every field's domain has a sort v that respects the universe bound and, where a large eliminator asks it, the subsingleton criterion (the binders of a reflexive field's own telescope respect the universe bound too);
- the constructor count respects the elimination rule;
- in the environment holding the type former and the constructors, the generated recursor's type has a type, and so has each rule's type, the recursor's binder prefix with the constructor's fields in place of the indices and the major.

Each “has a type” is an inference $\vdash T \Rightarrow S$ of §2, in the empty context, and each “has a sort” one whose result S reduces to a sort. So the generated types are checked like a definition's. The model needs only the inferences: the third claim of [Theorem 2](#) makes every generated type a well-denoted type.

IN CON-LECHE The same steps run in this order, and each generated rule is type-checked in the environment holding the rule-less recursor (a generated rule).

4.1.4. What is stored

The block adds three kinds of constant to the environment, the ones Section 3 left to this section (the kinds, as in con-leche). Written with the notation of Section 3, a stored declaration is one of

- $(c.\{\vec{p}\} : T := v)$, a definition (Section 3);
- $(I.\{\vec{p}\} : T)^{\text{type}}[k, m; c_1, \dots, c_n]$, a type former: k parameters, m indices, constructors $c_1, \dots, c_n$;
- $(c_j.\{\vec{p}\} : T)^{\text{ctor}}[I; k, f]$, a constructor of I : k parameters, f fields;
- $(r.\{\vec{\ell}\} : T)^{\text{rec}}[k, 1, n, m; R_1, \dots, R_n]$, a recursor: k parameters, one motive, n minors, m indices, the major; rules $R_1, \dots, R_n$, each flagged K when the block is K -like (Section 4.2.8).

The counts are the sizes of the argument groups ([Example 6](#) shows them on Nat). A rule R_j is a closed right-hand side over the recursor's level parameters, one per constructor. The rules are generated and stored without an inferred type: they mention the recursor itself, and the official kernel infers no rule either.

4.2. The model

The model of a block has three parts: the set the type former denotes at each choice of indices, the sets the constructors denote, and the set the recursor denotes. With the recursor, the contract of Section 3 gains its third law: every rule of every stored recursor satisfies its ι law ([Definition 9](#)), which this subsection proves.

Section 4.2.8 adds three more laws, facts about the model that let most ι steps go without a certificate. All of it is stated against an extension of §2's axioms.

4.2.1. The axioms, extended

Beyond the laws of Section 2.4 this section assumes that the positive universes $\mathcal{U}_n, n \geq 1$, are Grothendieck universes, and that there are tagged tuples:

1. Pairing, union, power set, replacement, on all sets: for sets a, b, x, A and a function F (of the meta-theory) from sets to sets, the sets $\{a, b\}, \bigcup x, \mathcal{P}(x)$ and $F[A]$ with

$$\begin{aligned} z \in \{a, b\} &\iff z = a \vee z = b, & z \in \bigcup x &\iff z \in y \text{ for some } y \in x, \\ z \in \mathcal{P}(x) &\iff z \subseteq x, & z \in F[A] &\iff z = F(w) \text{ for some } w \in A. \end{aligned}$$

2. The numerals and ω : a set $\text{num}(k)$ for every natural number k , distinct for distinct numbers, and the set ω of all numerals.
3. Grothendieck universes: for $n \geq 1$, $\mathcal{U}_n$ is transitive (if $A \in \mathcal{U}_n$ and $x \in A$ then $x \in \mathcal{U}_n$), closed under the four operations,

$$\begin{aligned} \{a, b\}, \bigcup x, \mathcal{P}(x) &\in \mathcal{U}_n && \text{for } a, b, x \in \mathcal{U}_n, \\ F[A] &\in \mathcal{U}_n && \text{for } A \in \mathcal{U}_n \text{ with } F(a) \in \mathcal{U}_n \text{ for every } a \in A, \end{aligned}$$

and contains ω .

4. Tagged tuples $\text{tag}(j, x_1, \dots, x_k)$, a list of sets with a number j attached: injective (equal tagged tuples have equal numbers and equal components), in $\mathcal{U}_n$ when every component is, never pt, and never a member of pt, so that a set of tagged tuples is strict.

Nothing else is assumed. There is no law about least fixed points and none about the size of an inductive family; both are theorems below. What the construction uses of the universes follows at once. A subset of a member is a member (it is in the power set, and the universe is transitive), so a separation from a member is a member; the union $\bigcup_{a \in A} F(a)$ of a family indexed by a member is a member, by replacement then union; and so is the countable union $\bigcup_k F(k)$, the union over ω . This is the one use of ω , and it is needed: the hereditarily finite sets satisfy every other law in this list.

IN CON-LECHE The set theory is one class: ZF without infinity, plus a chain of universes each of which is a Grothendieck universe stated as the matrix of Tarski's Axiom A with transitivity (the universe property, the class). The closure laws of the list, and infinity, are derived from it; the fragment takes the derived laws as its interface, as it does for the rest of §2's laws.

4.2.2. The operator

Fix a block as in Section 4.1, a valuation φ , and values $\vec{X}$ for the parameters. A *family* is a function W from index values $\vec{Y}$ to sets, its *fibres*. The block's operator Φ sends a family to a family: its fibre at $\vec{Y}$ is the set of *constructor values* whose fields fit a constructor relative to W and whose index expressions read as $\vec{Y}$,

$$\Phi(W)(\vec{Y}) = \left\{ \text{tag}(j, \vec{F}) \mid \vec{F} \text{ fit the fields of } c_j \text{ relative to } W, \text{ and } \vec{Y} = \vec{e}_j(\vec{F}) \right\},$$

where $\vec{e}_j$ are the index expressions of c_j 's result $I \vec{x} \vec{e}_j$ (Section 4.1), and $\vec{e}_j(\vec{F})$ their denotations with the parameters at $\vec{X}$ and the fields at $\vec{F}$. Values $\vec{F}$ fit a constructor's fields relative to W when each is a member of its field's set at the earlier values: the set of an ordinary field is its domain's denotation, read without W ; the set of a reflexive field $f : \forall \vec{z} : \vec{A}. I \vec{x} \vec{e}$ is the function space, read as in §2, over its telescope into fibres of W at the field's own index expressions $\vec{e}$,

$$\Pi^v(\vec{z} \in \vec{A}, \text{adm}(W(\vec{e}(\vec{z})))) ,$$

and the fibre $W(\vec{e})$ itself when the telescope is empty (a field's set, a fitting list). Here $\text{adm}(X)$, the admissible part of X , is X itself when the family is proposition-valued; when it is type-valued, it is the members of X that neither are the point nor contain it. On the family the construction ends with, adm changes nothing, since there a type-valued fibre is a set of tagged tuples. It is needed for the arbitrary families the operator is also applied to. A fibre $\{\text{pt}\}$ would make the field's function space a truth value, and then the field's set would not grow with the family (the function space's body). The fibre is a set, built by replacement, union and separation from the fields' sets (the construction, its members are as displayed). When the family is proposition-valued (u evaluates to 0 at φ) a constructor value is the point instead of the tagged tuple, so every fibre of $\Phi(W)$ is a subset of $\{\text{pt}\}$: the truth value “some constructor reaches these indices”.

4.2.3. Accessibility

Theorem 8 below, which supplies the closed family the least fixed point needs, asks for *accessibility*. An *occurrence* in a family W is a pair $(\vec{Y}, v)$ of index values and a member $v \in W(\vec{Y})$ of the fibre there; write $(\vec{Y}, v) \in W$. Call Φ *accessible with the bound* A , a set, when every element x of a fibre of $\Phi(W)$ has a *support*: a subset $B \subseteq A$ together with a map g that names, for each $a \in B$, an occurrence $g(a)$ in W , such that x is produced from every family in the universe that contains those occurrences:

$$\forall W \in \mathcal{U}_n, \vec{Y}, x \in \Phi(W)(\vec{Y}). \exists B \subseteq A, g. (\forall a \in B. g(a) \in W) \\ \wedge \forall W' \in \mathcal{U}_n. (\forall a \in B. g(a) \in W') \Rightarrow x \in \Phi(W')(\vec{Y})$$

($W \in \mathcal{U}_n$: every fibre of W is a member of $\mathcal{U}_n$). Whether an element is produced depends on at most A -many elements of the input and on nothing else, for the one set A . The fixed-point equation below also needs a monotone operator; accessibility implies it, on families in the universe: the support in W lies in any larger W' .

The block's operator is accessible. Take as an example the tree type with a constructor $\text{node} : (\text{Nat} \rightarrow \text{Tree}) \rightarrow \text{Tree}$ and an element $x = \text{tag}(j, f)$ of $\Phi(W)$: the field f is a function from the naturals into a fibre of W . Three things have to be shown.

1. *The support.* The value x depends on W only through the children $f \cdot 0, f \cdot 1, \dots$, so its support is those children: one occurrence for each argument z of the field, $(\vec{e}(z), f \cdot z)$, where $\vec{e}(z)$ are the field's index expressions. In general a reflexive field with telescope $\vec{z}$ contributes one occurrence per $\vec{z}$ fitting the telescope (a recursive field, with the empty telescope, exactly one), and an ordinary field contributes nothing. Each occurrence is named by the field's position and the tuple $\vec{z}$.
2. *A family holding the support produces x .* If every child lies in the matching fibre of W' , then f , a function whose applications all lie in fibres of W' , is a member of the function space into those fibres (by η), and ordinary fields do not mention the family; so $x \in \Phi(W')$.
3. *One bound for every family.* The names of the occurrences (field position and argument tuple) must come from one set A , the same whatever W is. For node that is the naturals. In general A collects, per constructor and reflexive field, the tuples fitting the field's telescope. A telescope may mention earlier fields, but by the second shape condition of Section 4.1 never a reflexive field's value, so the telescopes do not depend on W ; the bound reads them at the *one-fibre family*, every fibre the same one-element type: $\{\text{pt}\}$ when the family is proposition-valued, the singleton of a tagged tuple when it is type-valued (the replacement, the telescope reads alike).

Hence the operator is accessible with this bound, and so monotone.

For a type-valued family, that the bound is a member of $\mathcal{U}_{\varphi(u)}$, and that Φ maps families in $\mathcal{U}_{\varphi(u)}$ to families in $\mathcal{U}_{\varphi(u)}$, follows from the semantic form of the checker's universe bound, the universe bound on the fields: along any instance of any family in the universe, every ordinary domain and every entry of a reflexive field's telescope is a member of $\mathcal{U}_{\varphi(u)}$ (the bound is a member, the operator maps the universe to itself).

It comes from the checker's field-sort check, $v \leq u$, read in a model in which the type former denotes the one-fibre family: the type former's type

$$\forall \vec{x} : \vec{P}. \forall \vec{y} : \vec{J}. \text{Sort } u$$

is inhabited by the graph of any admissible family of types of the universe, so the environment holding the type former has a model for each one, and [Theorem 2](#) read in the one for the one-fibre family puts every field's domain in $\mathcal{T}_{\varphi(v)}$, and so, since $v \leq u$ and the universes are cumulative, in $\mathcal{U}_{\varphi(u)}$. No field reads an earlier reflexive field's value, so the domains met along an instance of any family are those met along an instance of the one-fibre family (the universe bound, at every family).

For a proposition-valued family none of this is needed. Every constructor value is the point, so every fibre of $\Phi(W)$ is a subset of $\{\text{pt}\}$, and the constant family $\{\text{pt}\}$ is closed under Φ and lies in $\mathcal{U}_0$. This closed family needs no bound in the universe and no universe bound on the fields. Accessibility, which holds in both cases, is used here only for monotonicity.

IN CON-LECHE The operator is read off the stored constructor types with the positivity check's variable (a *hole* standing for the family (the hole operator), and monotonicity and accessibility are read off the positivity check's run, case by case. A reading that mentions no hole is constant (the ordinary field); a function space over a hole-free domain, its fibres types, is positive when its body is and accessible with its body's bound glued over the domain (the reflexive field); a hole applied to hole-free arguments is positive and accessible with the bound $\{\text{pt}\}$ (the recursive field). Assembled along the fields, that is the fit's accessibility, for both kinds of family, with a bound in the universe for a type-valued one (the notion above), and the operator's monotonicity follows from it, as in the fragment. The fragment's simplification is to take the field kinds from the specification instead of the run. The universe bound on the fields the real proof records from a check made for the proof alone, the constructor type-checked with the holes in context (the recorded bound); the fragment reads it off the constructor's ordinary typing.

4.2.4. The least fixed point

A family L is closed under Φ , in $\mathcal{U}_n$, when every fibre of L is a member of $\mathcal{U}_n$ and $\Phi(L) \subseteq L$ fibrewise.

Theorem 8 (A closed family from accessibility). Let $n \geq 1$. An operator Φ on families that maps families in $\mathcal{U}_n$ to families in $\mathcal{U}_n$ and is accessible with a bound $A \in \mathcal{U}_n$ has a closed family in $\mathcal{U}_n$ (as in con-leche).

Proof. Iterate Φ along well-founded trees branching over all sets (a tree is a leaf, or a node with a subtree for every set). The *stage* at a leaf is the empty family, the stage at a node is Φ of the union over $a \in A$ of its subtrees' stages, and the closed family is the union of the stages over all trees. It is closed by accessibility: an element Φ produces from it has a support of at most A -many occurrences, each in some stage, and the node whose subtree at a is a tree of the occurrence $g(a)$'s stage produces the element at its own stage. It is in the universe because every stage is (Φ preserves the universe, which is closed under A -indexed unions), and because the union over all trees is a union over a *set*. A stage depends on its tree only through the tree's *code*, its set of finite paths over A , and the codes are members of the power set of all finite paths over A , itself a countable union of members of the universe (the code, the stage depends on the code only, the limit is in the universe). $\square$

With a closed family L_0 in hand, the least fixed point is a set construction, inside the set theory: fibrewise the intersection of the closed families, separated from L_0 ,

$$\text{lfp}(\Phi)(\vec{Y}) = \{x \in L_0(\vec{Y}) \mid x \in L(\vec{Y}) \text{ for every closed family } L\}.$$

Its fibres are members of $\mathcal{U}_n$, as separations from members; it lies below every closed family; and when Φ is monotone and maps $\mathcal{U}_n$ to itself it is a fixed point, $\Phi(\text{lfp}(\Phi)) = \text{lfp}(\Phi)$, and supports induction: a

property that holds of every element Φ produces from the family's separation by it holds on the whole family, because the separation is a closed family and the least one lies below it.

4.2.5. The family and the constructors

The block's operator has a closed family in $\mathcal{U}_{\varphi(u)}$: for a type-valued family by [Theorem 8](#), for a proposition-valued one the constant family $\{\text{pt}\}$ (the closed family). The family of the block is its least fixed point,

$$\llbracket I \rrbracket \cdot \vec{X} \cdot \vec{Y} = \text{lfp}(\Phi)(\vec{Y})$$

(the family, the type former's set, its graph over the parameters and indices). Its fibres are types of $\mathcal{T}_{\varphi(u)}$ (above), which is law 1 for the type former. A type-valued fibre is a set of tagged tuples, hence strict, and a proposition-valued fibre is a truth value, so either way the fibre is a type of the result sort. A member of a type-valued fibre is a tagged tuple and carries its constructor and its fields; a member of a proposition-valued fibre is the point and carries nothing.

The rest of this section rests on the fixed-point equation, read in both directions. A member of the fibre at $\vec{Y}$ is a constructor value whose fields fit the family and whose index expressions read as $\vec{Y}$, and conversely. The converse is law 1 for the constructors: c_j denotes the function that takes the parameters and the fields and returns $\text{tag}(j, \vec{F})$ (a constructor's set), a value in the fibre at its index expressions whenever its fields fit (the law). The forward direction is the *inversion* the ι law needs. The least fixed point's induction principle is induction over the family, on which the recursor is built. For a proposition-valued family every constructor value is the point, so the fibre is the truth value of “some constructor reaches these indices”, and every constructor denotes the point.

IN CON-LECHE The least fixed point is the same construction over a tuple of families, one per type former of the block, each a graph over an index set. What the model records of an installed block is one clause: the operator maps the universe to itself and its fit is accessible, for both kinds of family (the operator). So it has a closed family (the constant family at a proposition, and from accessibility for a type-valued family) and is monotone; its fibres are the tagged tuples of fitting fields (the fibres), and the type former denotes the least fixed point (the fixed-point equation). Its constructors are tagged pairs of nested pairs, the two kinds of family in one carrier.

4.2.6. The recursor

Fix values C for the motive and $\vec{S}$ for the minor premises. The recursor's value on a member of the family is determined by the rules. On $\text{tag}(j, \vec{F})$ it must be S_j applied to $\vec{F}$ and to the inductive hypotheses; at each reflexive field f the hypothesis is the function sending $\vec{z}$ to the recursor's own value at $f \cdot \vec{z}$, and at the empty telescope it is just the recursor's value at f . That this equation has exactly one solution is the *recursion theorem*. In the fragment it is proved by the same device as the family. The recursor's *graph*, the relation “the value at $(\vec{Y}, x)$ is v ”, is the least fixed point of the operator that reads the equation as a step, a least fixed point on predicates this time, which the ambient logic's impredicative Prop provides outright. It is single-valued by induction over the graph, using that tagged tuples are injective, and total by induction over the family.

When the family is proposition-valued and the motive is not, the major is the point and carries no fields. A major *decodes* as a constructor c_j with fields $\vec{F}$ when it is the value of c_j applied to fields $\vec{F}$ that fit c_j and whose index expressions read as the indices $\vec{Y}$; the pair is a *decoding* of the major. The point may have several decodings, and the equation has one instance per decoding. The graph is single-valued only because any two decodings agree: under a large eliminator the subsingleton criterion as the checker states it reads semantically as uniqueness of decodings, as [Theorem 10](#)'s proof shows.

The recursor denotes the graph of the resulting function (its value), curried over the parameters, the motive, the minors, the indices and the major. It is a member of its generated type, which is law 1 for the recursor (the recursor's typing, by induction over the family from the minors'). When the elimination level ℓ is zero at the valuation, the motive's values are truth values and the recursor's value is the point

at every argument. Then so is the recursor's set, since an abstraction is read by its values, and there is nothing to construct.

IN CON-LECHE The recursor is built the same way, except that its graph is also a least fixed point inside the set theory (the proof calls this the *graph route*), and one construction serves every sort. Over the set of majors, the graph is the least relation closed under the rules, each read over the decodings of a major. The graph has exactly one value at every major by the family's own induction, and the equation at any decoding is the ι law. The one fact about sorts it asks is a premise: at every major, two decodings are equal, or the motive's value there has at most one member. This is the elimination rule of Section 4.1, read three ways (the three cases, the theorem). At $\ell = 0$ the motive's values are truth values; at a type-valued family, tagged tuples are injective; at a proposition-valued family with a large eliminator, the subsingleton criterion makes the decoding a function of the index, which is the fragment's uniqueness of decodings as one premise.

4.2.7. The ι rule

The rule fires on $r.\{\vec{\ell}\} \vec{a} t$ where r is a stored recursor, $\vec{a}$ are its parameters, motive, minors and indices, and the major t reduces to a constructor application $c_j.\{\vec{\ell}'\} \vec{a}' \vec{f}$ for which r has a rule with right-hand side R_j . The reduct is $R_j[\vec{p}:=\vec{\ell}]$ applied to the parameters, motive and minors from $\vec{a}$ and to the fields $\vec{f}$ (as in con-leche). Besides finding r , c_j and the rule in the environment, the fragment's rule has three comparisons as premises:

- the constructor's levels $\vec{\ell}'$ are oracle-equal to the last levels of $\vec{\ell}$ (a large eliminator carries one extra level in front);
- the constructor's parameters $\vec{a}'$ are definitionally equal to the recursor's;
- the index expressions of the constructor's result type at $\vec{a}' \vec{f}$ are definitionally equal to the recursor's index arguments.

In one case it also has a *certificate* on the constructor application (Section 4.2.8). The official kernel makes none of these checks. The fragment's rules make the comparisons in every case except K, and its proof of the ι law uses them. The checker makes them only in the certified case of Section 4.2.8, together with the certificate; there they belong to the certification tax (Section 2.8).

Nothing checks the recursor's own arguments against its type, and nothing needs to. The reduction claim is asked only of a redex that is not the point, and such a redex fits the recursor's telescope as it stands. Call the arguments of an application its *spine*. Each prefix of the redex's spine is not the point either ($\text{pt} \cdot a = \text{pt}$), hence a graph. The condition the semantic invariant's application rule puts on the next argument names a function space holding that graph, the prefix's membership in its own type names another, and by domain uniqueness the two have one domain, which holds the argument.

IN CON-LECHE In the checker's rule every premise on the constructor application is conditional on the certified case. A K rule, and a family that is never a proposition, fire comparing nothing, as in the official kernel. The law there is proved without the comparisons: when the redex is not the point, the major is a tagged tuple, which ties the constructor application to the recursor's levels and parameters (what the law provides instead).

By Theorem 2 each comparison becomes an equality of sets wherever neither side is the point. So the ι law is stated on values, with the fits and these conditional equalities as premises:

Definition 9 (the ι law of a rule). Let r be a stored recursor and R_j the right-hand side of its rule for the constructor c_j . Take a valuation, levels $\vec{\ell}$ and $\vec{\ell}'$ of the right lengths, values for the recursor's arguments before the major (parameters $\vec{X}$, motive C , minors $\vec{M}$, indices $\vec{Y}$), and values for the constructor's arguments (parameters $\vec{X}'$ and fields $\vec{F}$). Suppose

- $\vec{X}, C, \vec{M}, \vec{Y}$ followed by $\llbracket c_j.\{\vec{\ell}'\} \rrbracket \cdot \vec{X}' \cdot \vec{F}$ fit the recursor's type at $\vec{\ell}$, and $\vec{X}', \vec{F}$ fit the constructor's type at $\vec{\ell}'$;
- $\vec{\ell}'$ evaluates as the last levels of $\vec{\ell}$;
- $X'_i = X_i$ at every position where neither is the point;
- the constructor's index expressions read at its arguments equal $\vec{Y}$ at every position where neither side is the point; and
- the recursor applied to all these values is not the point.

Then

$$\llbracket r.\{\vec{\ell}\} \rrbracket \cdot \vec{X} \cdot C \cdot \vec{M} \cdot \vec{Y} \cdot (\llbracket c_j.\{\vec{\ell}'\} \rrbracket \cdot \vec{X}' \cdot \vec{F}) = \llbracket R_j[\vec{p} := \vec{\ell}] \rrbracket \cdot \vec{X} \cdot C \cdot \vec{M} \cdot \vec{F},$$

and the right-hand side R_j is well-denoted with a well-formed application chain along those values.

The second conjunct makes the reduct well-denoted, as the first claim of [Theorem 2](#) demands; the first is the equation. The ι case of [Theorem 2](#) is then a translation. The comparisons become conditional equalities; the redex's own fit is as above; the constructor's fit comes from its certificate or from the arguments of [Section 4.2.8](#); the reduced major's value replaces the argument's; and the law gives the rest.

Theorem 10 (the ι law holds). Every rule of the recursor of an accepted block satisfies its ι law in the model of [Section 4.2](#). (Via the equation on the semantic recursor; in con-leche, the graph's equation, lifted to the law of [Definition 9](#).)

Proof. First the conditional comparisons become equalities. The recursor's and the constructor's telescopes are generated from one parameter context and one index context, and both spines fit them. Under equal earlier values, two values compared at one position lie in one type, whose members are all the point or none, so where the comparison says nothing they are both the point (a conditional comparison of fitting values is an equality). So $\vec{X}' = \vec{X}$, and the constructor's index values are $\vec{Y}$.

The type-valued family. The recursor's fit puts the major's value $\llbracket c_j \rrbracket \cdot \vec{X} \cdot \vec{F}$ in the family at the recursor's parameters $\vec{X}$ and indices $\vec{Y}$, and by the constructor's fit that value computes to $\text{tag}(j, \vec{F})$. A member of the fibre is a constructor value of fitting fields (the fixed-point equation, read forwards), and tagged tuples are injective, so the fields fit c_j 's field telescope at the recursor's parameters, which is what the β steps inside R_j require. The recursion theorem's equation at $\text{tag}(j, \vec{F})$ is the rule's equation.

The proposition-valued family. Here the major's value is the point, and the recursor's fit says only that the fibre at the recursor's indices is inhabited. The point carries no constructor, no fields, no parameters and no indices. When the elimination level is zero the recursor's application is the point, and the law asks nothing. Otherwise the block passed the subsingleton criterion. It has one constructor, so every decoding of the point is by c_j , and each field is a proposition or occurs among the constructor's result indices. A propositional field is the point in both decodings, provided both read its domain at one valuation of the block's level parameters; the level comparison secures that. The index equality identifies an index field's value with the recursor's index argument. So the two decodings of the point agree, and the recursion theorem's equation is again the rule's. $\square$

Why the index comparison is needed. Take the recursor of [Example 7](#) on the spine $C, h, 7, \text{mk } 5$. Every $P\ n$ is inhabited, by $\text{mk } n$, so $\llbracket P\ 7 \rrbracket = \{\text{pt}\}$, and the recursor's fit asks only that $\llbracket \text{mk } 5 \rrbracket = \text{pt}$ lie in it, which it does. The fit holds equally for $\text{mk } 7$. Without the index comparison the ι law would therefore have to give both

$$\llbracket P.\text{rec} \rrbracket \cdot C \cdot h \cdot 7 \cdot \text{pt} = h \cdot 5 \quad \text{and} \quad \llbracket P.\text{rec} \rrbracket \cdot C \cdot h \cdot 7 \cdot \text{pt} = h \cdot 7,$$

so $h \cdot 5 = h \cdot 7$ for every h , which is false. The recursor's set is a function of its arguments and has one value at index 7. On values the types $P\ 5$ and $P\ 7$ are both $\{\text{pt}\}$, so no comparison of types could tell them apart; the index comparison, $5 \equiv 7$, is what the model can use. For a type-valued family the comparison

is redundant, because the tagged tuple carries its indices. The official kernel never makes it: at an ι step it compares nothing and relies on the term being well-typed, which its typing judgement guarantees. Our proof has no typing judgement, so where the major may be the point the comparison has to be a premise of the rule.

4.2.8. When ι needs a certificate

The official kernel checks nothing at an ι step: the term is well-typed, so the constructor application it reduces to and that application's fields are well-typed too. The semantic proof needs the reduct to be well-denoted and the constructor's arguments to fit its type wherever the redex is not the point. What supplies that depends on the major's family at the instance. The model has three more laws for it, each proved when the block is installed.

A family that is never a proposition. If the family's sort is at least 1 at every valuation (one oracle test on the levels at the use), then the step needs no certificate. The model's *fibre law* says that a stored family's fibres are truth values where its stored sort evaluates to zero and strict sets where it does not; the installation proves it from the construction of Section 4.2. The redex is not the point, so the major lies in its domain, a strict fibre. So the major is not the point, its own reduction claim applies, and the constructor application it reduces to is not the point and lies in the same fibre. Hence it fits the constructor's telescope by the argument of Section 4.2.7 (the case's proof). This covers every redex on `Nat`, `List`, `Bool` and the like.

A family that may be a proposition. This is the certified case. The rule makes the three comparisons of Section 4.2.7 and checks the constructor application: the constructor's levels, parameters and index expressions are compared with the recursor's, and every parameter and field of the constructor application is inferred by full inference and compared with the constructor's domain. At a valuation where the family is a proposition the constructor's value is the point (a constructor of a family in Prop), and an application whose head is the point constrains none of its arguments. So a field can lie outside its domain in a well-denoted major. The reduct applies the minor premise to that field, and is then not well-denoted although the redex is (a witness, in which the minor is a graph over `{pt}` and the field is not the point). This is the second kind of check of the certification tax (Section 2.8). Other checkers skip it for the reason they skip the first: in a well-typed major the fields are well-typed. The major premise itself needs no check. By the model's *major-premise law*, wherever the recursor's domain holds the point as the major, the redex or the constructor is the point (its proof: a fibre holding the point is a truth value, so the family is a proposition there). In that case the reduced constructor application is the point, as the major is.

K. A block whose sort is `Prop`, with one constructor, which has no fields (`Eq`, `HEq`, `True`), is *K-like*, and its rule is flagged. Its step fires on any major whose type is right, with no constructor in sight. The reduct is the rule's right-hand side at the recursor's parameters, motive and minors alone. There is no certificate and no comparison, and the major is not reduced. What remains is Lean's type check: the major's inferred type reduces to the family at some arguments, and the constructor fabricated at their parameters has a type definitionally equal to it. That check decides whether the step is taken; no claim consumes it. The model's *K law* says that at *any* values fitting the recursor's whole telescope, the major's included, an application that is not the point equals the right-hand side at the values before the indices. The installation proves it by inversion of the least fixed point. The major's value lies in the fibre at the recursor's parameters and indices, a member of a fibre is a constructor value reaching those indices, and with one constructor and no fields that value and those indices are determined. The proof uses neither the family's sort nor the elimination level, so the law holds for every one-constructor block without fields; the `Prop` condition is Lean's.

For a structure, Section 6.2 shows a further way to fire ι without a certificate, through the projections of the major.

IN CON-LECHE The checker decides the case with one level test per redex: whether the constructor's family is never a proposition at the levels the rule compares with, read off the sort recorded for the family when it was

installed. A K rule and a family that is never a proposition fire with no check; every other rule is the certified case, as in the fragment.

4.3. Consistency

An environment is *accepted* when it is built from the empty environment by the two steps of §3 and §4: a definition that passes its checks, or an inductive block that passes its checks and is installed. An accepted environment is closed: every stored term mentions only stored constants. This is what [Theorem 5](#) and the construction of Section 4.2 assumed of the environment they extend.

Theorem 11 (Every accepted environment has a model). For every model of the extended theory, every accepted environment has a model in it (con-leche’s main theorem).

Proof. By induction on the acceptance. The empty environment has a model; a definition step is [Theorem 5](#); a block step is the construction of Section 4.2, *assembled*: law 1 is its membership claims, law 2 has no new instance, and law 3 is [Theorem 10](#). $\square$

IN CON-LECHE The block step is one theorem, the install of a block, read off the checker’s run stage by stage.

Corollary 12 (No proof of an empty proposition). Let I be an inductive type with no constructors, no parameters, no indices and no level parameters, installed in an accepted environment. Then no closed term e has $\vdash e \Rightarrow I$; in particular no stored constant has type I (at the block inductive `False : Prop`; as in con-leche).

Proof. Take a model of the environment, by [Theorem 11](#), and read law 1 for the block’s recursor at the valuation that sends every level to 0. Its type is then

$$\forall (C : \forall (t : I). \text{Prop}). \forall (t : I). C \ t,$$

the elimination principle of an empty type, read as a proposition, and law 1 puts the recursor’s set in what it denotes. If e had $\vdash e \Rightarrow I$, [Corollary 3](#) would put $\llbracket e \rrbracket$ into $\llbracket I \rrbracket$. Take for C the abstraction over $\llbracket I \rrbracket$ that is constantly the empty truth value: it lies in the motive’s domain, so the recursor applied to C and $\llbracket e \rrbracket$ lies in $C \cdot \llbracket e \rrbracket$, the empty truth value: a member of the empty set. The proof uses only that a model exists, not how it was built. (The construction of Section 4.2 also makes $\llbracket I \rrbracket$ empty outright, as the least fixed point of an operator with no step.) $\square$

IN CON-LECHE The real theorem differs in two ways. Its checker does not read `False` from the stream but installs it from a built-in copy and rejects a stream that declares it differently (likewise `Eq`, `Nat` and a few others, Section 7), so the main theorem can name `False` and say that it denotes the empty set and `Eq` set equality, with no hypothesis about the stream ([§1 of OVERVIEW.md](#)); and its acceptance is the run of a program, the declaration fold, rather than a relation, so a bridge theorem turns each accepting run into the derivations §3 and §4 consume (Section 7). The argument in between is the one above.

5. Nested inductive types

This section changes tack. The previous sections present a fragment of the checker and prove it sound. This one describes in full con-leche’s check for *nested* blocks, which mention the type being defined inside another inductive type (mutual blocks aside), and then only sketches how the construction and the proofs of Section 4.2 extend to them. The details are in the linked con-leche proof. A restricted case, with one container instantiation at depth one and not under a binder, is verified in the paper’s Lean fragment: installing such a block keeps a model of the environment.

5.1. Nested blocks

A block of Section 4 mentions the type being defined only directly: a recursive field's domain is the family itself, a reflexive field's a function type into it. The standard example of such a nested block is a tree whose children form a *list* of trees:

```
inductive Tree (α : Type) where
| node : α → List (Tree α) → Tree α
```

The domain of the field `List (Tree α)` is the block `List`, installed earlier and called the *container*, applied to the type being defined. The type `Tree α` there is the *nested occurrence*, and `List (Tree α)`, the container at given levels and parameters, is an *instantiation* of the container.

The container may also sit under a binder, as in

```
inductive P where
| mk : (Nat → List P) → P
```

and nesting may go deeper. With lists, rose trees and

```
inductive L (α : Type) where
| nil : L α
| cons : α → L α → L α
inductive R (α : Type) where
| node : α → L (R α) → R α
inductive T where
| leaf : T
| mk : R T → T
```

the block `T` is nested through `R`, a container that is itself nested: inside `R T`, the constructor `R.node` holds an `L (R T)`, so `T` occurs at depth two. Containers with indices are allowed too, as in

```
inductive TV (α : Type) where
| node : α → {n : Nat} → Vec (TV α) n → TV α
```

with `Vec α n` the vectors of length `n`.

5.2. The positivity check

Section 4.1 classifies each field by the shape of its domain as ordinary or reflexive. The checker does this with one walk over the constructors, the same for every block, which also looks through containers. On a block of §4's shape only the first three cases below apply, and they give exactly §4's classification.

Holes and frames. The walk works on variables in place of types. Before a constructor is examined, every application of the type being defined to the block's parameters, $I \vec{x}$, is replaced by a variable X , a *hole*, standing for that whole application (a family over the indices). For this to catch every occurrence, the walk is preceded by a *uniformity check*: the type being defined occurs in a constructor only applied to exactly the block's parameters, at the block's own levels, and in no parameter's domain. A *frame* is one type whose constructors are being walked, at one instantiation. It is either the block itself (the *root frame*) or a container instantiation met in a field, such as `List at Tree α`. A container's frame has holes of its own: in `List`'s constructors the application `List α` of the container to its own parameter is replaced by a hole Y *before* α is instantiated, so it is not confused with an application at another instantiation. Frames nest: walking one may meet another container, whose frame is walked inside it. At any point the *types in progress* are the type being defined and every instantiation whose frame is open, each present as its hole.

The cases. The walk first puts a field's domain in weak head normal form w , and every case reads w , never the domain's syntax, so a container reached only by unfolding a definition counts as one. Inside the current frame, the cases are these.

1. *Ordinary.* If w mentions no type in progress, the field is *ordinary*.

2. $A \forall$. If w is $\forall(z : A). B$, the domain A must not mention a type in progress, or the block is rejected as not positive, and the walk continues with B , z a new variable. The binders passed form the field's telescope.
3. *A hole*. If w is a hole applied to arguments, the arguments (the indices of the type the hole stands for) must all be present and must not mention a type in progress, or the block is rejected. The root frame's hole makes the field *recursive* (*reflexive* when a $\forall$ was passed). The hole of a container frame, this one or an enclosing one, stands for an instantiation in progress, such as the tail of a `List.cons`, and is accepted.
4. *A container*. If w is a stored inductive type C at levels $\vec{u}$ applied to arguments, where C is neither one of the block's types nor the quotient type, the first k_C arguments (C 's parameter count) are its parameters $\vec{D}$, the rest its indices. It is required that
 - the indices mention no type in progress (they may mention earlier fields, as the n of `Vec (TV  $\alpha$ ) n` does);
 - the parameters mention no variable but the block's parameters and the holes, so no field and no variable bound by a $\forall$ of the field;
 - $\vec{u}$ has C 's number of levels, and every index of C is supplied;
 - C 's index telescope at $\vec{D}$ mentions no type in progress, and C 's sort is equivalent to the block's.

The parameters may mention the types in progress anywhere, as in `List (Nat  $\rightarrow$  Tree  $\alpha$ )`; the frame decides whether that is positive. Then the instantiation $C.\{\vec{u}\} \vec{D}$ is looked up. If it is in progress, it is rejected: its syntactic occurrences have become its hole, so meeting it as a type here means reduction produced it. Otherwise its frame is walked, as described next, unless the same instantiation was accepted before. The field is then *nested*, under the $\forall$ s it passed.

5. *Anything else*, such as the type being defined at other levels, a variable that is not a hole, or a quotient, is rejected.

Walking a frame. At an instantiation $C.\{\vec{u}\} \vec{D}$, the walk checks that $C.\{\vec{u}\} \vec{D}$ has a type, and introduces a hole for it: a variable standing for the container's type former applied at this instantiation. It then takes each of C 's constructors in turn:

- its stored type at $\vec{u}$, the frame's applications replaced by the holes and the parameters instantiated at $\vec{D}$, must be a type;
- every field is classified by the cases above, inside this frame;
- no later field and no result index may depend on a field that is not ordinary (§4's second shape condition, at the instantiation);
- the result's indices mention no type in progress;
- the constructor's walked type is recorded, every hole read back as the application it stands for, for the recursors below.

The root frame is the same loop over the block's own constructors.

On `Tree`, the field of type `List (Tree  $\alpha$ )` meets the container case at `List` with the parameter X . In `List`'s frame, `List.nil` has no field, and `List.cons` has the fields $h : X$, the root's hole, which is recursive, and $t : Y$, the frame's own hole, which is accepted. In `P.mk : (Nat  $\rightarrow$  List  $P$ )  $\rightarrow$  P` the walk passes a $\forall$ over `Nat` and meets `List` at X the same way. For the rose trees, `T.mk`'s field meets R at X ; in R 's frame, with its hole Y_R , `R.node`'s fields are X and $L Y_R$, so L 's frame opens inside R 's, with its hole Y_L , and there `L.cons`'s fields are Y_R and Y_L , both accepted. A container that uses its parameter to the left of an arrow, such as $C \alpha$ with a constructor of type $(\alpha \rightarrow \text{Nat}) \rightarrow C \alpha$, fails at the instantiation: in its frame that field's domain is $X \rightarrow \text{Nat}$, a $\forall$ whose domain mentions the hole. Whether a container is positive in its parameter is thus decided at the instantiation, by walking it; nothing about it is recorded when the container is installed.

IN CON-LECHE The official kernel translates a nested block into an auxiliary mutual block, with a copy of the container per instantiation; con-leche builds no auxiliary type, walks the container at the instantiation (§5 of [OVERVIEW.md](#)), and records the instantiations it accepted and, where it can, does not walk an accepted one again.

5.3. Classes and their recursors

A recursion over a tree has to pass through the list of its children. So a nested block's recursors eliminate several types, its *classes*. They are the type being defined and the container instantiations the recursion passes through: every instantiation at which a field of the block or of a walked container lands, read with its holes back as the types they stand for, is a class. For `Tree` the classes are `Tree  $\alpha$` and `List (Tree  $\alpha$ )`; for the rose trees, `T`, `R T` and `L (R T)`. Each class gets a motive and a recursor (`T.rec` for the type being defined, `T.rec1`, `T.rec2`, ... for the others), and each constructor of each class a minor premise. All recursors share one prefix, made of the parameters, the motives and the minor premises. A minor premise is generated as in Section 4.1 from the constructor's fields at the instantiation, with one inductive hypothesis per recursive or nested field: the motive of the class where the field lands, under the field's $\forall$ s, read off the walk's recorded type (a recursor's type, a rule).

Example 13 (Tree's recursors). With C the motive for `Tree  $\alpha$` and C_1 the motive for `List (Tree  $\alpha$ )`, the shared prefix is the parameter, the two motives, a minor premise for `Tree.node` and one for each of `List`'s constructors at the instantiation:

$$\begin{aligned} \text{Tree.rec}.\{\ell\} : & \forall(\alpha : \text{Sort } 1). \forall(C : \forall(t : \text{Tree } \alpha). \text{Sort } \ell). \\ & \forall(C_1 : \forall(ts : \text{List (Tree } \alpha)). \text{Sort } \ell). \\ & \forall(s : \forall(a : \alpha). \forall(ts : \text{List (Tree } \alpha)). \forall(h : C_1 \text{ } ts). C (\text{Tree.node } a \text{ } ts)). \\ & \forall(n : C_1 \text{ List.nil}). \\ & \forall(c : \forall(t : \text{Tree } \alpha). \forall(ts : \text{List (Tree } \alpha)). \\ & \quad \forall(h : C \text{ } t). \forall(h_1 : C_1 \text{ } ts). C_1 (\text{List.cons } t \text{ } ts)). \\ & \forall(t : \text{Tree } \alpha). C \text{ } t \\ \text{Tree.rec}_1.\{\ell\} : & \dots \forall(ts : \text{List (Tree } \alpha)). C_1 \text{ } ts \end{aligned}$$

(the $\dots$ is the same prefix; the constructors' own parameter arguments are left implicit). The minor for `Tree.node` has one inductive hypothesis, C_1 at the nested field; the minor for `List.cons` has two, C at the field that held the hole X and C_1 at the one that held Y . With $\vec{r}$ for the prefix $\alpha \ C \ C_1 \ s \ n \ c$, the three rules are

$$\begin{aligned} \text{Tree.rec } \vec{r} (\text{Tree.node } a \text{ } ts) & \rightsquigarrow s \ a \ ts \ (\text{Tree.rec}_1 \ \vec{r} \ ts) \\ \text{Tree.rec}_1 \ \vec{r} \text{ List.nil} & \rightsquigarrow n \\ \text{Tree.rec}_1 \ \vec{r} (\text{List.cons } t \text{ } ts) & \rightsquigarrow c \ t \ ts \ (\text{Tree.rec } \vec{r} \ t) \ (\text{Tree.rec}_1 \ \vec{r} \ ts). \end{aligned}$$

A nested field under binders gets its hypothesis under the same binders: for $P.\text{mk} : (\text{Nat} \rightarrow \text{List } P) \rightarrow P$ the minor premise is

$$\forall(f : \text{Nat} \rightarrow \text{List } P). \forall(h : \forall(n : \text{Nat}). C_1 (f \ n)). C (P.\text{mk } f).$$

Where a rule of $T.\text{rec}_1$ fires. A rule of Section 4.2 takes the constructor's levels and parameters to be the recursor's own. At a container class they are not: `List.cons` in the class `List (Tree  $\alpha$ )` has the parameter `Tree  $\alpha$` , not the recursor's α , and `List`'s level names, not `Tree`'s. So each rule of a recursor at a container class stores the instantiation, that is, the class's levels and parameters as terms in the recursor's level parameters and parameters (the stored rules at a container class). In the certified case of ι (Section 4.2.8), the constructor application's levels and parameters are compared with the stored ones, instantiated at the recursor's levels and parameter arguments (the levels by the level oracle, the parameters definitionally), and the index comparison is as before. In the other cases the rule fires comparing nothing, as for a simple block. The constructor's fields are what follows its own parameters.

Large elimination. A nested block may eliminate into every sort only if its sort is never zero (where the checker enforces it), as in the official kernel. A nested proposition eliminates only into `Prop`, and the subsingleton criterion is never asked of a nested block.

IN CON-LECHE As for a simple block (Section 4.1), the classes and the layout of the prefix are read off the stream's recursor types by an unverified pre-pass, and each class is then checked: the type being defined at the block's levels and parameters, or a stored inductive type, not the quotient, whose parameters mention the type being defined and nothing but the block's parameters, in the block's sort. Every class other than the type being defined is walked as a container met at the root, so every class is an instantiation the walk visited. A field landing at a class for which the stream gives no recursor is rejected. One class may be visited at several places. In a block with a field of type $\text{List } (RL \text{ TL})$, with $RL \alpha$ a rose tree over lists, that instantiation is met at the block's field and again inside $RL \text{ TL}$, and the generator checks that every visit gives each field the same inductive hypothesis.

5.4. The model, as a sketch

What follows is intuition for how the construction of Section 4.2 carries over. The details are in the linked con-leche proof.

The operator. Fix the parameters, and let W be the *approximant*, the family the operator is applied to, as in Section 4.2. A container field ranges over the container's own family at the instantiation, with W in place of the type being defined: for Tree , the field $\text{List } (\text{Tree } \alpha)$ of Tree.node ranges over the lists whose entries all lie in W . That family is known, because every installed block leaves a clause in the model: its type former denotes, at all parameter values, the least fixed point of its operator. So the container's least fixed point is reused, at each approximant, and nothing new is constructed for it. At depth two the same holds one level down: an element of $R \ W$ holds a list from L 's family at the parameter $R \ W$.

Accessibility. The block's operator stays accessible, because the walk that checked positivity also bounds the support. An element of the container's family at W depends on W only through the positions the walk found: the fields that held the root's hole, such as the entries of a list, and, through the fields that held the frame's own hole, the same positions of its sub-values. Read as an operator of two arguments, the approximant and the container's own family, the container's operator at the instantiation is accessible by the same case-by-case argument as §4's, run on the frame; and the least fixed point of such an operator is accessible in its first argument, the supports of an element's sub-values collected along paths into one support. Monotonicity follows from accessibility as in §4. For the recursor, con-leche also proves the walk's readings monotone case by case; at a container that compares the container's least fixed points at two instantiations, by leastness. Nothing about the container's parameter is recorded when it is installed. The walk at the instantiation is what makes the argument go through.

The family. The container lives in the block's sort (the container case of Section 5.2 requires it), so its family at an approximant in the universe is in the universe, and the operator maps the universe to itself. With accessibility, Theorem 8 gives a closed family, and the block's family is the least fixed point as in Section 4.2.4, with its fixed-point equation and its induction principle: a property that holds of every element the operator produces from elements having it holds of the whole family.

The recursors. For a simple block, Section 4.2 defines the recursor's value by the recursion equation and obtains it as a *graph* in the set-theoretic sense: a relation between elements of the family and values, the least one closed under the equation, which is then shown to be a function. A nested block has several recursors, one per class, and they call each other: the ι rules of Section 5.3, read as equations, define Tree.rec on Tree.node in terms of Tree.rec_1 at the list of children, and Tree.rec_1 on List.cons in terms of Tree.rec at the head. So the two are defined together, by one relation. It is the least relation between a class (here $\text{Tree } \alpha$ or $\text{List } (\text{Tree } \alpha)$), an element of that class, and a value, closed under all these equations at once. As in Section 4.2 it is single-valued, because tagged tuples are injective. At the elimination level zero both sides are the point, and a large eliminator out of a proposition is refused for nested blocks, so the decodings argument of Section 4.2 is not needed. It is total (every element of every class has a value) by one induction over all the classes together: the induction principle of Tree 's family, which at the nested field $\text{List } (\text{Tree } \alpha)$ hands over to the induction principle of List 's family at that instantiation, and so on as deep as the walk went (the induction over the classes, read at the recursors' classes). Each recursor is then this relation restricted to its class, carried over its arguments.

The $\mathbf{t}$ laws. They follow as in Section 4.2. For a rule of `Tree.rec1` the major is a constructor of `List` applied; the recursor's fit puts its value in the class at the recursor's parameters, and `List`'s fixed-point equation there identifies the constructor and the fields with the rule's (the graph's equation, lifted to the law).

6. Consequences of extensionality

The official kernel has two rules of definitional equality that the fragment does not have: η for structures, and unit-likeness. Section 7 lists them among the omissions. This section shows that adding either would cost one case of the induction of Section 2.7 and no new idea. In the model, each rule follows from what the values are. A proof is the point, a function is the graph of its applications, a constructor value is a tagged tuple of its fields, and a member of an inductive family is a value that one constructor step produces. Each of these says that a set is determined by its parts, and each is an extensionality law of the assumed set theory or follows directly from one. η for functions, which the fragment has, is the same law at graphs: a member of a function space is the graph of its applications, and §2 proved its case. The K rule of Section 4.2.8, which the fragment also has, rests on a law of the same kind, inversion of the least fixed point.

Recall what Section 4 sets up. A constructor application $c \vec{p} \vec{f}$ of a block whose family is type-valued denotes a tagged tuple $\text{tag}(i, \llbracket f_1 \rrbracket, \dots, \llbracket f_n \rrbracket)$, the constructor's number i followed by its fields. The family at parameters and indices denotes the least set closed under the constructor steps, so that a member of the family is a tagged tuple that one constructor step produces from members of the field domains. This is the fixed-point equation of §4, read from left to right. When the family is proposition-valued (its result sort `Sort  $u$` has u evaluating to 0 at φ), the family denotes a truth value instead. The constructor step's tuple is not stored, only whether some such tuple exists, and a member of the family, like a constructor application, is the point. Tagged tuples are injective (tuples, tags) and a tagged tuple is $\neq \text{pt}$.

6.1. Proof irrelevance and propositional extensionality

These two need no rule; they are built into the model, and §2 used the first already. A proposition denotes a truth value, a subset of $\{\text{pt}\}$, so any two proofs of any two propositions denote the same set. That is the proof-irrel case of [Theorem 2](#), which never compared the two propositions. And two propositions that imply each other have the same truth value, by extensionality of sets. This is Lean's axiom `propext`, which therefore holds in the model.

6.2. η for structures

The rule. A *structure* is a block with one constructor `mk`, no indices, no recursive field, and a type-valued family. The official kernel equates any s of the structure type with the constructor applied to s 's projections: $s \equiv \text{mk } \vec{p} s.1 \dots s.n$. The check is that s 's type reduces to the structure at the parameters $\vec{p}$, and compares each field of the constructor application with the corresponding projection of s . The fragment has no projection terms (Section 7). Read $s.i$ below as the projection defined through the recursor, `S.rec  $(\lambda \vec{f}. f_i)$   $s$` , whose value is the i -th component of the tuple by the $\mathbf{t}$ law, or as a primitive projection, whose interpretation reads the component directly. The argument is the same either way.

Lemma 14 (structure η). Let s denote a member of the structure's family at $\vec{p}$, and let f_i denote the i -th component of that member's tuple for each i . Then $\llbracket \text{mk } \vec{p} f_1 \dots f_n \rrbracket = \llbracket s \rrbracket$.

Proof. By the fixed-point equation, the member $\llbracket s \rrbracket$ is a value one constructor step produces: $\llbracket s \rrbracket = \text{tag}(0, x_1, \dots, x_n)$ for some $x_1, \dots, x_n$ in the field domains. There is only one constructor, so the tag is 0. Tagged tuples are injective, so the i -th component of $\llbracket s \rrbracket$ is a function of $\llbracket s \rrbracket$, and it is x_i ; that is what f_i denotes. The constructor application denotes $\text{tag}(0, \llbracket f_1 \rrbracket, \dots, \llbracket f_n \rrbracket) = \text{tag}(0, x_1, \dots, x_n)$, the same set. $\square$

The lemma proves nothing specific to the structure. The equation is what membership in the family means, read from left to right, plus the fact that a tuple determines its components. In the soundness case the checker’s field comparisons deliver $\llbracket a_i \rrbracket = \llbracket s.i \rrbracket$ for the fields a_i the constructor was actually applied to, and the lemma does the rest. A block declared in `Prop` is never granted the rule. Where an instance of a `Sort u` structure happens to be a proposition, both sides are the point, and the law’s proposition case covers it.

ι through the projections. The same law on values, that a member is the constructor at its projections, gives a second way to fire ι on a block of a structure’s shape (one constructor, no indices, no recursive field). For such a block whose instances may be propositions, ι can go through the projections of the major instead of a constructor application:

$$S.\text{rec } \vec{p} C m x \rightsquigarrow m (S.p_1 \vec{p} x) \dots (S.p_k \vec{p} x).$$

This step is sound with no certificate, no comparison and no reduction of the major, given the plain ι law and the structure- η law on values. The redex is not the point, so the whole spine, the major included, fits the recursor’s telescope (Section 4.2.7). Where the structure is a proposition the major is the point, and so are its projections when every field’s sort is `Prop` wherever the structure’s is; the reduct is then the minor at the point, which is the recursor’s value there. The step must expand the major as found, before any reduction of it, and also when it already is a constructor application. A major that denotes the point may reduce to a term that is not well-denoted, and the projections of that reduct break the reduction claim (the witness). For a structure without fields the step is K’s step (the reducts coincide). The fragment has no projections, so this step is a theorem about the claims, and not a rule of its relations.

IN CON-LECHE The checker has the rule as Lean has it, the constructor’s fields compared with the projections of the other side, with its soundness case. It uses the same law when it prepares the major premise of a recursor on a structure that is never a proposition at the instance. The major is taken as found, before any head normalisation, and the constructor is applied to its projections, with no further check (the rule, its soundness case). Only if that does not give a constructor application is the major head-normalised, and the same expansion is tried once more on the result. Whether a type has the rule at all is decided once, when it is installed, from its shape (one constructor, no index, not a proposition, in a block where no constructor is recursive), and recorded with the type; the law itself is established there too, not at the use. It is the η law of the structure’s *projection table*, the record of its fields’ types that the checker stores for every one-constructor, index-free type (Section 7, “Projections”) (a member is the constructor at the parameters and its own projections). It is proved when the checker stores the table, by the argument above on the tagged tuple.

6.3. Unit-likeness

The rule. A block with one constructor, no indices and no fields, such as `PUnit` or, at `Prop`, the proposition `True`, has one element up to definitional equality: the checker equates any two terms whose types reduce to it.

Lemma 15 (unit-likeness). If s and t denote members of the family of such a block, at the same parameters, then $\llbracket s \rrbracket = \llbracket t \rrbracket$.

Proof. When the family is type-valued, Lemma 14 with $n = 0$: each member is `tag(0)`, the one tagged empty tuple. When it is proposition-valued, the family is a truth value and both members are the point. $\square$

A set with at most one member takes one of two shapes in the model: the singleton of the empty tagged tuple for a type-valued family, and a truth value for a proposition-valued one. The lemma holds for both by the same argument.

IN CON-LECHE The checker has the rule for any stored unit-like family, with its soundness case. Unit-likeness is recorded at installation, like η (one constructor, no index, no field, in a block where no constructor is recursive),

and the law is established there from the fixed point (the fibre is the one tagged empty tuple). Neither `PUnit` nor `True` is special to the checker: both are installed like any other block.

7. What we left out

The fragment of §2–§4 is the real checker with everything removed that does not change the shape of the argument. This section lists what was removed, with one sentence on what the real proof does about it and a link to the code. Then it describes where the real proof’s architecture differs from the fragment’s on the features they share.

7.1. Dropped features

Projections — Lean’s primitive projection `e.i`, the i -th field of a structure value. For every inductive type with one constructor and no index the checker stores a projection table, the fields’ types read off the constructor. The real checker reduces a projection of a constructor application to that field and infers its type from the table; in the model a structure value is the tagged tuple of Section 6.2 and a projection reads a component, which the install of the table establishes (its typing and ι laws).

Mutual inductive types — Several types declared together, each constructor free to mention any of them. The real checker checks such a block the same way as a single type, with a tuple of operators, one per type of the block, and each type’s family is a component of their simultaneous least fixed point. The argument is the one for a single type, with more indices, so this document skips it.

Nat and String literals, and the fast Nat path — Numerals and strings are terms of their own; the checker expands a literal to its constructor form when reduction needs it (`natLit`, `strLit`) and folds `Nat.succ` and the binary `Nat` operations on literals with machine arithmetic. The folding is sound only for the operations as the toolchain defines them, so the checker compares each stream’s definition against a pinned copy and the model establishes the operations’ recurrences from that comparison.

Quotients — `Quot`, `Quot.mk`, `Quot.lift`, `Quot.ind` and the axiom `Quot.sound` are installed from a built-in pin and modelled by the set-theoretic quotient of a set by a relation, for which the pinned constants are shown to be members of their types.

Pinned blocks — In the fragment `False` is whatever empty proposition the stream declares. The real checker installs `Eq`, `Nat`, `Empty`, `False` and the quotient `Quot` from hand-written copies of the toolchain’s declarations (`False`, for instance, the builder): a stream block that carries a pin’s names and agrees with it up to renaming of universe parameters installs the pin, and one that disagrees is rejected. The pins are how the main theorem can name `False` and `Eq` and say what they denote.

Axioms — The fragment has none. The real checker accepts exactly Lean’s three standard axioms: `propext` and `Classical.choice`, each pinned to the toolchain’s statement and each true in the model (`propext` by the extensionality of truth values, choice by choice in the meta-logic), and `Quot.sound` as part of the pinned `Quot` block above. Any other axiom record declines the stream, with two tolerated exceptions: a declared but unused `sorryAx` installs nothing, and Lean’s deprecated compiler-trust axioms are accepted as pinned definitions of their own types.

Theorems and opaques — The fragment has definitions only. The real checker never unfolds a theorem or an opaque: only a definition unfolds.

Structure η and unit-likeness — The checker has structure η and unit-likeness as Lean has them; Section 6.2 and Section 6.3 derive both from extensionality.

Recursors on stuck proofs of $A \wedge B$ — The checker never unfolds a theorem, while the official kernel does. Some elaborated terms rely on that, with a case split on a stuck proof h of $A \wedge B$. There the recursor fires on `And.intro h.1 h.2`, the constructor applied to h ’s two projections, built from h ’s type.

Fuel, memo tables, the executable checker and its bridge theorem — In the fragment the relations are the checker. The real checker is a program, connected to the relations by further theorems (Section 7.2).

The two-phase parallel fold — The fold installs every declaration first and then checks the recorded declarations one by one, each against the prefix of the environment it was installed at; the binary runs the second phase on a pool of workers whose results are reassembled in record order, so its verdict is the fold’s. The proof reads an accept of the fold as an environment in which every declaration was checked where it was installed, which is what the one-declaration-at-a-time argument of §3 and §4 needs.

7.2. How the real proof differs from the fragment

Free variables carry their types. The paper uses named variables and a context. The real checker keeps no context at all. When it opens a binder it substitutes a free variable that carries its own type, `fvar_idx type`, identified by the depth at which it was opened; so a variable is never looked up, the relations are indexed by the opening depth in place of a context, and two free variables are compared by depth alone. Stored types are closed terms, and `Denotes`, the relation the main theorem is stated over (below), reads them with de Bruijn indices under binders, with no opening at all.

An intermediate term language. The real proof does not interpret the checker’s terms directly. It first reads a term into a smaller language, `Term`: de Bruijn indices, sorts at concrete numbers (the level valuation already applied), and built-in constants at concrete levels. That reading, `denote`, is partial only for syntactic reasons: an unknown constant, a wrong number of levels, or a projection with no table. The interpretation `interp` then maps the term to a set, reading each binder off the values as Section 2.5 does. The paper folds the two steps into one denotation on the terms themselves.

Six relations, and a mode index. Beside reduction, definitional equality and inference the real description has three more: the certification of an argument spine against a binder telescope (the certified case of ι), pairwise definitional equality, and the congruence of two lambda telescopes (Section 2.8). A rule with a premise about a list or a telescope needs a relation of its own in a mutual block; the paper inlines them into the rules that use them. Inference also carries a mode index, for `infer_only` (Section 2.9), which the fragment states as a relation of its own.

The bridge and the recomposition. Because the real checker is a program, its proof has three parts where the paper has one. The bridge turns an accepting run at any fuel into a derivation, by one induction on fuel in which each checker site lands on one rule. The soundness is the paper’s induction: one structural induction over the six relations, each case a lemma about one rule, never mentioning the implementation. The recomposition puts them together, claim by claim. It bridges the run, applies the soundness, and hands the result to the declaration fold.

The statement over `Denotes`. The main theorem is stated over `Denotes`, a relation on the checker’s own terms with one rule per syntax form, written to be read in one sitting, and over the structure `Model` built on it. A theorem connects the two: wherever the reading `denote` exists, its interpretation is a `Denotes`-denotation of the term. The paper’s theorem is the model side directly.

The cached checker and its simulation. The core the binary runs is a clone of the pure one over a term representation that carries a precomputed hash in every node, with memo tables for reduction, equality and inference. A simulation proof shows that every successful run of the cached core is a successful run of the pure fuelled core at some fuel, with the memo tables’ invariant preserved; the main theorem composes it with the fold’s soundness.

The frontend. Between the export file and the declaration fold sit a decoder and one transformation of the decoded list, `preparePrelude`. It moves the declarations of a built-in copy of the toolchain’s prelude (Eq,

`Nat`, `Bool` and their kin) to the front, the stream’s own record where it has one and the copy’s where it has none, and it moves the structural `Nat` operations that a pinned `Nat` operation is defined from ahead of it. A theorem says that this is all. The fold’s input is a permutation of the decoded list plus records of the built-in prelude, and nothing is rewritten or generated. What no theorem says is that the decoded list means the same as the export file: the decoder is written to be meaning-preserving, and that is a review claim. The fragment has no frontend: its stream is its environment.

8. Related work

This section compares our proof with a small selection of the most closely related work.

Carneiro, “The Type Theory of Lean”. The thesis [1] presents Lean’s type theory in full by a typing judgement, studies its metatheory, and builds a set-theoretic model, from which Lean’s consistency follows relative to ZFC together with, for each n , the existence of n inaccessible cardinals. It is the starting point of our model, whose interpretation follows the thesis’s. A proof is one point, a proposition $\forall x : \alpha. \beta$ is the intersection of its fibres with the singleton of that point (our propositional function space Π_0 , Section 2.5), and a $\forall$ into a higher universe is the full dependent function space. To decide which reading applies, the thesis first translates every well-typed term into a language in which the propositional and the non-propositional $\forall$ and lambda are separate constructs, using the sort of each body, which is well defined given the thesis’s unique typing theorem. Our interpretation reads the choice off the values instead (Section 2.5): a function space is a proposition when all its fibres are truth values. So it consults no typing information, and our proof does not need unique typing. The thesis models a fixed basis of eight inductive types, the W -type among them, to which it reduces the others (parts of the reduction left as future work); here every inductive block the checker accepts is interpreted by its own least fixed point (Section 4.2). The statements also differ in form. There, each derivation of a formal typing judgement is sound in the model once a number of inaccessible cardinals depending on the derivation is assumed; here the theorem is about the verdicts of the checker’s own rules, and assumes the whole chain of universes $\mathcal{U}_0, \mathcal{U}_1, \dots$ at once (Section 2.4).

Lean4Lean. Lean4Lean [2] is a checker for Lean 4 written in Lean, closely following the official kernel and able to check all of Mathlib. The paper also formalises a typing judgement, a slight variation of the thesis’s, and proves the main body of the checker correct against it: an equality test that answers yes yields a definitional equality of the judgement, an inferred type a typing; inductive types and additions to the environment are listed as unfinished. The paper notes that this correctness proof needs, in some form, unique typing and the inversion of definitional equality on sorts and $\forall$ -types for its judgement. Our proof relates the checker’s verdicts to the model directly, with no typing judgement in between, and uses no such property (Section 2.7). The Lean development `lean4lean-model` [3], built on Lean4Lean’s judgement, states as its target theorem the consistency of that judgement assuming ω inaccessible cardinals, a proof still to be supplied at the time of writing; the set theory `con-leche` assumes follows from that hypothesis.

Werner, “Sets in types, types in sets”. Werner [4] interprets the Calculus of Inductive Constructions in ZFC with an increasing sequence of inaccessible cardinals, one for each universe, and conversely encodes ZFC in the type theory, so that the two hierarchies interleave in strength. The interpretation is proof irrelevant: a proposition denotes 0 or $1 = \{0\}$, a proof denotes 0, a $\forall$ into propositions denotes 1 exactly when every fibre does, and a function type of the predicative levels denotes the full set of set-theoretic functions. These are the two readings of a $\forall$ in Section 2.5. It is defined on a term in its context, by structural induction on the term, and partially: whether the clause for proofs or for propositions applies to a term is a matter of its type. It is shown defined and sound on every derivable judgement by induction on the derivation, and the case of the conversion rule uses subject reduction and confluence of reduction. Our interpretation is total and term-directed, reads the choice off the values, and [Theorem 2](#) is an induction over the checker’s three relations that uses neither. Werner interprets an inductive type of a universe as the least fixed point, in that universe, of the monotone operator that strict positivity provides, and his

calculus has no inductive propositions; here proposition-valued families are least fixed points too, in $\mathcal{U}_0$ (Section 4.2).

Barras, “Sets in Coq, Coq in Sets”. Barras [5] axiomatises intuitionistic ZF, possibly with inaccessible cardinals, inside Coq, develops functions, ordinals and fixpoint theory over the axioms, and proves set-theoretic models of the Calculus of Constructions sound, as well as its extension with a hierarchy of universes (interpreted by Grothendieck universes) and an extension with the natural numbers. The set theory is given by axioms, as ours is (Section 2.4). Π -types are interpreted by one clause: with Aczel’s encoding of a function as the set of pairs (x, y) with y in its value at x , a function into propositions collapses to a proposition, so the interpretation, like ours, consults no typing information. In our set theory a graph is never the point (Section 2.4), so that a graph determines its domain; the two readings of a $\forall$ or a lambda are then different sets, and the values choose between them (Section 2.5). Soundness there is stated for semantic judgements $\Gamma \models M : T$ and $\Gamma \models M = M'$ over valuations, close to our three claims, and the typing derivations of the calculus, in a presentation with judgemental equality, are shown to imply them; our claims are proved of the checker’s verdicts instead. Barras observes that set-theoretic β holds only for arguments in the function’s domain, so his semantic β rule has a typing premise; in our proof β has no premise, because the reduction claim is asked only of a redex that is not the point, whose function is then a graph that determines its domain (Lemma 1).

Dybjer, “Inductive families”. Dybjer [6] gives a general scheme for inductive definitions of families of sets in Martin-Löf’s type theory: formal criteria for the formation and introduction rules of a new set former (parameters, indices, non-recursive premises and recursive premises), and an inversion principle that derives the elimination and equality rules from them. A recursive premise may be a function over a sequence of types that does not mention the set former being defined (strictly positive, generalised induction). Our blocks follow this scheme (Section 4.1): ordinary fields are his non-recursive premises, reflexive fields his recursive premises with the field’s telescope as that sequence, and recursive fields the case of an empty one. His order, non-recursive premises before recursive ones, appears here as the condition that nothing after a reflexive field depends on its value. The recursor’s type and its ι laws are generated from the constructors as his elimination and equality rules are. In our proof they are statements about the model, where the recursor’s set is constructed and the ι law proved (Theorem 10). Proposition-valued families, and the subsingleton criterion for their large elimination, belong to the impredicative setting, which his scheme, for a predicative theory, does not have.

Abbott, Altenkirch and Ghani, “Containers”. Abbott, Altenkirch and Ghani [7] represent strictly positive types as containers: a set of shapes and, for each shape, a set of positions, an element being a shape with an element of the argument at each of its positions. They show that nested strictly positive inductive and coinductive types exist in any Martin-Löf category (locally cartesian closed, with disjoint coproducts and W-types) by representing them as containers and constructing them from W-types. In these terms an element’s support (Section 4.2) is the set of its positions with the occurrences that fill them, and the bound A is one set that holds the positions of every shape. Our proof represents no operator as a container: the support and the bound are read off the block’s fields, and a nested occurrence is handled by checking the container’s own constructors at the given instantiation (Section 5.2) rather than by composing containers. Coinductive types are outside our proof, and indexed families outside the paper’s containers.

Traytel, Popescu and Blanchette, “Foundational, compositional (co)datatypes for higher-order logic”. Traytel, Popescu and Blanchette [8] build datatypes and codatatypes in Isabelle/HOL, with mutual and nested recursion and corecursion, from *bounded natural functors*: type constructors equipped with a map function, set functions returning the elements an argument holds, and an infinite cardinal bounding those sets, subject to conditions including naturality and the bound. The class of these functors is closed under composition, initial algebras and final coalgebras, so a datatype nested through `list` uses the fact, proved once, that `list` is a bounded natural functor; it includes non-free type constructors such as finite sets. The bound is what lets the initial algebra be constructed inside HOL. Our proof uses no map function and records nothing per type former: the operator’s accessibility, a support bound in the same role, is read off the checker’s positivity check for each block, and a container’s positivity in its parameter is decided at

each instantiation by walking its constructors (Section 5.2), never stored. Coverage differs as well: their framework has codatatypes and nesting through non-free type constructors; ours has indexed, universe-polymorphic families and proposition-valued ones, and no codatatypes.

Aczel, “An introduction to inductive definitions”. Aczel’s chapter [9] treats inductive definitions by rule sets and by monotone operators on the subsets of a set: the set inductively defined is the least set closed under the rules, which is the least fixed point of the operator and is reached by iterating it. It calls an operator κ -based when every element it produces from a set X it already produces from a subset of X of cardinality less than κ (as when every rule has fewer than κ premises), and shows that for regular κ such an operator’s iteration closes by stage κ . Accessibility (Section 4.2) is this idea of rules with a bound on their premises, the bound being a set A indexing the support rather than a cardinal. What is specific to our proof is where the bound comes from. It is read off the checker’s positivity check for each block, with one occurrence per reflexive field and tuple of its telescope, and, for a nested block, at each instantiation of a container (Section 5.2).

References

- [1] M. Carneiro, “The Type Theory of Lean,” Master’s thesis, Carnegie Mellon University, 2019. [Online]. Available: <https://github.com/digama0/lean-type-theory/releases/tag/v1.0>
- [2] M. Carneiro, “Lean4Lean: Verifying a Typechecker for Lean, in Lean,” in *31st International Conference on Types for Proofs and Programs (TYPES 2025)*, in Leibniz International Proceedings in Informatics (LIPIcs), vol. 384. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2026, p. 2:1–2:23. doi: [10.4230/LIPIcs.TYPES.2025.2](https://doi.org/10.4230/LIPIcs.TYPES.2025.2).
- [3] M. Carneiro, “lean4lean-model.” [Online]. Available: <https://github.com/digama0/lean4lean-model>
- [4] B. Werner, “Sets in Types, Types in Sets,” in *Theoretical Aspects of Computer Software (TACS ’97)*, in Lecture Notes in Computer Science, vol. 1281. Springer, 1997, pp. 530–546. doi: [10.1007/BFb0014566](https://doi.org/10.1007/BFb0014566).
- [5] B. Barras, “Sets in Coq, Coq in Sets,” *Journal of Formalized Reasoning*, vol. 3, no. 1, pp. 29–48, 2010, doi: [10.6092/issn.1972-5787/1695](https://doi.org/10.6092/issn.1972-5787/1695).
- [6] P. Dybjer, “Inductive Families,” *Formal Aspects of Computing*, vol. 6, no. 4, pp. 440–465, 1994, doi: [10.1007/BF01211308](https://doi.org/10.1007/BF01211308).
- [7] M. Abbott, T. Altenkirch, and N. Ghani, “Containers: Constructing Strictly Positive Types,” *Theoretical Computer Science*, vol. 342, no. 1, pp. 3–27, 2005, doi: [10.1016/j.tcs.2005.06.002](https://doi.org/10.1016/j.tcs.2005.06.002).
- [8] D. Traytel, A. Popescu, and J. C. Blanchette, “Foundational, Compositional (Co)datatypes for Higher-Order Logic: Category Theory Applied to Theorem Proving,” in *27th Annual IEEE Symposium on Logic in Computer Science (LICS 2012)*, IEEE, 2012, pp. 596–605. doi: [10.1109/LICS.2012.75](https://doi.org/10.1109/LICS.2012.75).
- [9] P. Aczel, “An Introduction to Inductive Definitions,” in *Handbook of Mathematical Logic*, J. Barwise, Ed., North-Holland, 1977, pp. 739–782. doi: [10.1016/S0049-237X\(08\)71120-0](https://doi.org/10.1016/S0049-237X(08)71120-0).