BendTT: An Affine Dependent Type Theory

Victor Taelin
Higher Order Company
Rio de Janeiro, Brazil
taelin@higherorderco.com

AI Disclosure. Bend and BendTT were designed by the human author. This paper was written by Claude Fable 5.1 from the author's code and design choices, and reviewed by the author. The Lean mechanization was human-specified, AI-proven, and verified by a computer. Human paper soon™.

Abstract

BendTT is the type theory of the Bend programming language. It has one sort with Type : Type, no universe hierarchy and datatypes with no positivity restriction, and it is consistent. What holds it up is a usage discipline: a value is consumed at most once unless the *kind* of its type says otherwise. A binder marked `+` may be consumed any number of times, and it forms only at the kind Data. A function type is never Data, and a datatype earns Data at every constructor. So no closure is ever copied, and every known paradox of Type : Type or of negative datatypes copies a closure. Recursion passes one syntactic descent test. Erased code is free and may diverge; nothing promotes it to live. Proofs are ordinary definitions: the match is the eliminator and there are no tactics. We state the calculus, show how each attack dies, and describe the core's Lean 4 mechanization.

1 Introduction

Bend is a functional language with dependent types and a parallel runtime [25]; BendTT is the theory its checker implements. It keeps Type : Type, impredicative quantification and recursive datatypes with negative occurrences, and stays consistent. This paper explains why.

The engines of the classical paradoxes all copy a function: Girard's paradox and its Hurkens form apply a function-typed value to itself [15, 17], and Curry's paradox through a negative datatype applies a node's field to a copy of the node [10]. Logics without contraction admit naive comprehension [16, 26]. BendTT carries that idea into a dependent type theory, where it replaces the universe hierarchy and the positivity check: a live variable is consumed at most once, and a function is never an exception.

Affinity alone would make the language useless: a proof feeds one hypothesis to the induction hypothesis and to a lemma. Bend restores reuse through the kind of a type. A binder may be marked `+`, and then its type must have kind Data. A datatype declares its kind and the checker earns it at every constructor; a function type is Type; an equation is Data because its evidence is erased. The license to copy is a property of a *type*, never a proof carried by a *term*: there is no modality, no copy class, no clone function. This inverts quantitative type theory [3, 4, 20], where ω is the ordinary binder and consistency comes from the universe hierarchy the theory keeps. Here 1 is the ordinary binder and ω is a permission a type earns. One rule shows the difference: an argument that enters a `+` binder is counted once, where QTT scales its usage by ω; Section 3.4 says what that costs. Everything else is cheap: one bidirectional pass [12] with a usage counter per binder, no unification, one syntactic test per self-call, and a dead fragment that costs nothing and is erased before the runtime.

t, A, B ::=

`x` , `k`	variable, reference to a definition
`@q x: A -> B`	function type
`x => f` , `f(a)`	abstraction, application
`q x = v`	let; the body follows
`Kind(g)` , `Quant`	a kind, the sort of quantities
`&0` `&1` `&2` , `g <&> h`	quantity literals, the meet
`D<p..>` , `C{a..}`	family instance, constructor
`\{C: h; m}` , `\{}`	one-constructor match; empty match
`{a == b : T}` , `{==}`	propositional equality, reflexivity
`%e : P`	rewrite by e through P; the body follows
`{x : T}` , `?name`	annotation, hole

Figure 1. Terms, in the surface syntax. q is a quantity sigil: `-` for 0, none for 1, `+` for ω. A family D carries an unspellable set r of already-matched constructors. Kinds are terms: Type = Kind(1) and Data = Kind(2).

2 The Calculus

2.1 Terms, Quantities, Kinds

Figure 1 lists the term formers; types are terms. A binder carries a quantity $q \in \{0, 1, \omega\}$, spelled `-x`, `x` and `+x`: erased, affine, reusable. Every type has a *kind* Kind(g) where g is a quantity term: a literal, a variable of the sort Quant, or a meet $g \sqcap h$ (surface `<&>`). Conversion orders kinds by the quantity order, Kind(g) $\leq$ Kind(h) when $h \leq g$: Data fits every kind, every kind fits Type, a kind fits a meet when it fits either side, a meet fits a kind only when both its sides do, and a stuck quantity fits only itself. The meet reduces only when forced: ω is its identity, 0 absorbs, two literals meet, and a stuck side stays stuck.

A *book* is an ordered list of declarations. A `type` declares a family with parameters and a kind, then its constructors, each a telescope of fields tipped at the family. A `law` declares a name at a closed type and a later `def` fills it. Until its fill a name is an *axiom*: it may appear in types and other dead positions, and live code may not consume it. A definition may reference itself only through the descent rule of

```
(x : q´A) ∈ Γ
──────────────────────────────────────────── var
Γ ⊢q x : A ▹ {x ↦ q}

book(k) : T
q = 1 ⇒ k is filled, and a self-call descends
──────────────────────────────────────────── ref
Γ ⊢q k : T ▹ ∅

Γ ⊢₀ A : Kind(q´)     Γ, x : q´A ⊢₀ B : Type
──────────────────────────────────────────── all
Γ ⊢q ∀q´x:A. B : Type ▹ ∅

Γ, x : q´A ⊢q f : B ▹ π      π(x) ≤ q´
──────────────────────────────────────────── lam
Γ ⊢q λx. f : ∀q´x:A. B ▹ π \ x

Γ ⊢q f : ∀q´x:A. B ▹ π     Γ ⊢q″ a : A ▹ π´
q″ = 0 if q´ = 0, else q
──────────────────────────────────────────── app
Γ ⊢q f(a) : B[x ≔ a] ▹ π + π´
```

```
Γ ⊢₀ A : Type    Γ ⊢₀ a : A    Γ ⊢₀ b : A
──────────────────────────────────────────── eql
Γ ⊢q {a == b : A} : Data ▹ ∅

C ∉ r      q = 1 ⇒ q´ ≠ 0
Γ ⊢q h : ∀(qᵢ·q´)xᵢ:Fᵢ. P(C{x₁…}) ▹ π
Γ ⊢q m : ∀q´s:D^(r∪{C})<p…>. P ▹ π´
──────────────────────────────────────────── mat
Γ ⊢q \{C: h; m} : ∀q´s:D^r<p…>. P ▹ π ⊔ π´

q = 1 ⇒ q´ ≠ 0
D^r is empty, or a live (x : E) ∈ Γ with E empty
──────────────────────────────────────────── efq
Γ ⊢q \{} : ∀q´s:D^r<p…>. P ▹ ∅

Γ ⊢q e : {a == b : A} ▹ π
Γ ⊢₀ P : ∀x:A. {a == x : A} → Type    P(b, e) ≤ T
Γ ⊢q f : P(a, {==}) ▹ π´
──────────────────────────────────────────── rwt
Γ ⊢q (%e : P) f : T ▹ π + π´
```

Figure 2. Typing, the rules that differ from the textbook. `Γ ⊢q t : T ▹ π` reads: under the book and context Γ, at demand $q \in \{0, 1\}$ (0 dead, 1 live), t has type T and consumes π, a map from variables to quantities. $\pi + \pi'$ adds pointwise ($1 + 1 = \omega$), $\pi \sqcup \pi'$ takes the maximum, and $r \cdot q'$ is 0, q' or $q' + q'$ as r is $0, 1$ or ω. In MAT, F_i and q_i are the constructor's fields at the family's parameters. Conversion, constructors and family instances are textbook; a let `q´ x = v` over b is $(\lambda x.\, b)(v)$ with v inferred and its type checked dead at $\mathsf{Kind}(q')$.

Section 2.5, and never a later name, so the reference graph is acyclic and mutual recursion cannot split a loop across two definitions. The flattener compiles `match` / `case` blocks into a case tree of one-constructor peels [19], uncovered rows becoming the empty match.

2.2 Reduction and Conversion

Evaluation is weak head reduction. A lambda applied steps; a let substitutes; a reference unfolds only when its spine reaches its declared arity and it has a body, so an axiom and an underapplied definition are stuck values; a match meeting its constructor passes the fields to the arm, and any other constructor falls whole to the default m, whose domain records the peeled constructor in r; a rewrite sheds when its evidence reaches `{==}`. When a definition's match sticks on a variable, the evaluator answers the definition applied to that variable, not the exposed case tree, so goals stay in the vocabulary of the source; this refolding is what lets an induction hypothesis match a goal (Section 4).

Conversion $A \leq B$ is reduce-and-compare at every node, up to η for functions, and it is a preorder, not an equality: directional at kinds and at match residuals (more constructors peeled fits fewer), domains of a function type swapped, quantities exact, and every part that flows both ways (an argument, a parameter, a field, an equation endpoint) compared as a symmetric $A \equiv B$. Reflexivity uses $\equiv$: `{==}` does not prove `{Data == Type : Type}`, which J could transport into a cast. Conversion may diverge on dead code (Section 3.4).

2.3 Typing

Figure 2 gives the rules; each is also a derivation comment beside its case in `bend2/bend.ts`. Inference synthesizes, checking pushes a goal into the introduction forms, and the two meet at conversion. Both directions carry a *demand* q: 0 checks a term dead, 1 live. There is no demand ω: a term checked at demand ω would let a binder inside it contract, and Atkey shows this also breaks substitution [3]. Sequential premises add their measures, the arms of a match join since one of them runs, and a binder validates $\pi(x) \leq q'$ when it closes: two live uses of an affine binder saturate to ω and fail there. A reference costs nothing: code is free.

The dead fragment is free. Types in premises, binder domains, equality endpoints and rewrite motives check at demand 0, where the measure is dropped. A dead term may mention consumed variables, may diverge, and may inhabit `Empty`. No rule coerces dead to live: an erased binder's uses count at 0, and a match on an erased scrutinee is refused in a live region.

Reuse is gated, not scaled. An argument to a binder of quantity q' checks dead if the binder is erased and at the ambient demand otherwise, and its measure adds once. So a value bound once may enter a `+` binder, and the callee copies it: *certify-once*. It is licensed because the `+` binder's domain checked against `Data` when the function type formed, and a `Data` value holds nothing affine (Section 2.4). A match hands each field out at the field's quantity times the scrutinee's, so a `+` scrutinee makes its plain fields reusable.

Matching is consumption. A match is a value of function type, and applying it consumes the scrutinee. The arm receives the fields at a goal specialized to the rebuilt constructor, $P(C\{x_1 \ldots\})$: dependent elimination with no generated eliminator and no unification. The default receives the scrutinee with C peeled onto r, and the empty match closes the chain when no constructor remains, or when a *live* binder

in scope has an emptied type; an erased one proves nothing, since dead code inhabits it.

Equality is the $\mathcal{J}$ axiom with an explicit motive. From $e : \{a == b : A\}$, a rewrite maps a goal $P(b, e)$ to the obligation $P(a, \{==\})$; the programmer writes P, and it checks dead. The evidence runs, so it checks at the ambient demand. An equation is Data: its evidence is erased, and a closed live proof normalizes to `{==}`, so copying it copies nothing.

2.4 Kinds Are Earned

A `+` binder forms only over a Data type, so the security of the theory rests on what may be Data. The rules assign kinds by shape: a function type is Type, because a closure captures; an equation is Data; a family has the kind it declares, Kind(G) over its parameters. The book validator earns the declaration: for each constructor it walks the telescope, parameters then fields, in the real constructor context, and checks a binder of quantity q against Kind(q) and an affine field against the declared Kind(G); the tip must be the family applied to its own parameters, in order. From the base library:

```
type List<a, -A: Kind(a)> is Kind(a):
  Nil{}
  Con{head: A, tail: List<a, A>}
```

A list is as reusable as its element, and the dependent pair `Sigma<a, b, A, B>` is `Kind(a <&> b)`: reusable when both halves are. `Nat`, `Bool`, `U32` and `String` are Data; `Array`, the IO handles and the effect type are Type. A recursive occurrence is assumed at the declared kind, with no fixed point: a live value is a finite tree, so the invariant that a Data value holds nothing affine follows by induction on the value. Values never weaken with their kinds: `List<&2, Nat>` and `List<&1, Nat>` are different types. Generic code takes the quantity as an erased parameter (`for -a: Quant`), under which `+` is refused, since `Kind(a)` does not reduce to Data; the license appears only after instantiation.

2.5 Descent

Self-reference passes one test, run against the definition's own case tree. Walking the body, the checker rebuilds the definition's left-hand side: a lambda binds the next *column*, a match peels the current column into a constructor of fresh field columns. At a live self-reference the pending arguments compare against the columns left to right, erased columns skipped: each must equal its column until one is a *strict subterm* of it, the same constructor with a strictly smaller field, or a term sitting inside one of the pattern's fields. Arguments past the decreasing one are free, so Ackermann passes: its inner call shrinks the first column, its outer call keeps it and shrinks the second. The comparison sees through lets, so `+p = p0` keeps `p` a subterm. No sizes are computed: this is a minimal member of the structural-recursion family [14], chosen because it is one pass and trivial to audit. Dead demands skip the test, so a type may recurse freely; that is what makes `R = @-x: R -> Empty` definable, and it is harmless because dead code never runs.

3 Why It Is Consistent

There are two ways to loop: self-application and recursion. Descent closes the second. This section is about the first, and why the copy license cannot be forged. Every example is a test in the repository, and the messages are the checker's own.

3.1 Self-Application Dies at the Counter

$\Omega = (\lambda x.\, x\, x)(\lambda x.\, x\, x)$ uses its binder twice, so the measure saturates to ω, and a plain binder refuses it. The only binder that admits two uses is `+x`, and `+x` forms only over a Data type. No function type is Data:

```
law dupf:
  for +f: Nat -> Nat
  Nat
```

```
- expected : Data
- observed : Type
```

Hurkens' paradox [17] is twenty definitions long in Bend and typechecks up to `tauinner`, the first place a function-typed variable is used twice; it dies there with `f (consumed more than once)`. Impredicativity and Type : Type do no harm on their own.

3.2 Negative Types Are Legal, and Harmless

There is no positivity check. A HOAS-style term type is a fine declaration, and its evaluator is a fine program:

```
type Trm is Type:
  Lam{f: Trm -> Trm}
  Num{n: U32}

def app(t: Trm, x: Trm) -> Trm:
  match t:
    case Lam{f}:
      f(x)
    case Num{n}:
      Num{n}
```

Curry's engine, `omega(Lam{f}) = f(Lam{f})`, needs the field `f` twice and dies at the counter like `dupf`. A `Data` declaration cannot hold the function either: a field of type `Trm -> Trm` under `is Data` fails with the same error. So a negative field sits only in an affine type, where it is used at most once, and a value of a negative type is at worst an inert closure.

3.3 The License Cannot Be Forged

Every attack we know of tries to obtain Data for something affine. The *copy paradox* is the sharpest: in a theory whose contractible types are reusable, the type of copies of x, `&y: T -> {x == y : T}`, has one inhabitant, `(x, {==})`, and copying that pair copies x. Here it must be a datatype with a declared kind, and its live field of type T has kind Type, which Data does not admit: the declaration fails at its constructor. A constructor that binds its own quantity `L: Quant` and stores a field at `Kind(L)` fails the same way, since a local `L` is never assumed to be `&2`. A *dead hypothesis* licenses nothing: an erased `-e: Empty` or `-c: Copiable(T)` in

scope may never be supplied, and a `+` needs a kind that *reduces* to Data, which a type stuck on dead evidence never does. A *quantity equation* `{&1 == &2 : Quant}` is a legal type with no live proof, and a rewrite through it yields a value whose type is a stuck rewrite that nothing applies. The meet charges both operands, so `q <&> &2` is not a free copy of `q`; `@x: A -> B` and `@-x: A -> B` are different types at conversion; and a circular fill of `Forge(T): Data` is a live self-call with no shrinking column, refused by descent.

3.4 The Claims, and Their Price

For a book that validates, the theory claims subject reduction, progress, weak normalization of closed live terms, and no closed live inhabitant of `Empty`. The last three are stated at the live demand because dead code may rest on an axiom, diverge, or inhabit `Empty` by design: erased code is specification, not proof. Conversion, and so the checker, is a semi-decision procedure, as it already is in theories with divergent types [9]: a hang is "inconclusive", never "accepted". The stance is soundness on accept, not checker totality, in the line of NuPRL and Zombie [6, 8].

Certify-once has a price. Term-substitution reduction does not preserve the usage measure: unfold `f(+x)` at `f(y)` and `y` counts twice under its plain binder. So subject reduction for usage is claimed for weak by-value reduction of closed terms, the only reduction the machine performs: when `f` unfolds, `y` is a value whose resources were consumed once, and the copy is Data, which owns nothing. Three invariants outside the checker carry this. No pass duplicates a term: the evaluator shares every argument, let value and field in a memoized cell, and the compiler is strict. A type with runtime ownership (a file, a socket, an array) is declared Type, never Data; this is a property of the base library, not a rule. A compiler may drop a copy the source spelled, never add one [25]. QTT avoids the question by scaling the argument's measure, which it can afford because ω is its default; with 1 as the default, scaling would leave only closed data reusable.

One escape hatch exists and is always disclosed. A definition marked `@unsafe` skips descent and forms its `+` binders at any kind; the checker reports every book that uses one, so a clean report means none of the claims above is waived.

4 Proofs Without Tactics

A claim is a `law` and a proof is the `def` that fills it. `for` folds to a function type, `exs` to a dependent pair, `where` packs a hypothesis onto a binder; `{a != b : T}` is a function into `Empty`.

The match is the eliminator. Matching a parameter refines the claim in each arm: the goal in the `1n+p` arm of a claim about `a` is the claim at `1n+p`, evaluated, with stuck self-calls refolded to source form. The induction hypothesis is the recursive call, and descent makes it valid. With no unification and no metavariables, a match scrutinizes only a parameter or a field, so the law of a helper *is* the motive of its match: a computed scrutinee goes to its own definition. Rewriting is J with the motive written out: given `e : {a == b : T}`, the motive marks with `_` the places where `b` stands, the goal must be the motive at `b`, and the body proves it at `a`. Commutativity of addition, from the lemmas `zero : {a == add(a, 0n)}` and `succ : {1n+add(a, b) == add(a, 1n+b)}`, each by the same induction:

```
def add(a: Nat, b: Nat) -> Nat:
  match a:
    case 0n:
      b
    case 1n+p:
      1n+add(p, b)

law comm:
  for +a: Nat
  for +b: Nat
  {add(a, b) == add(b, a) : Nat}

def comm(a, b):
  match a:
    case 0n:
      zero(b)
    case 1n+p:
      %succ(b, p) : {1n+add(p, b) == _ : Nat}
      %comm(p, b) : {1n+add(p, b) == 1n+_ : Nat}
      {==}
```

The successor arm uses `p` twice and `b` twice. `Nat` is Data, so `+a` and `+b` license it, and a `+` scrutinee hands out `+` fields. This is the idiom that affinity alone forbids and the kind restores. Statements are free: at demand 0 a theorem may quantify over functions and repeat variables at will. Affinity constrains what runs, never what is said.

5 Erasure and the Runtime

The checker elaborates as it checks and hands the compiler the term with every node typed [25]. Erasure drops erased binders, arguments and fields, and every type, kind, quantity, equation and proof; a rewrite compiles to its body. What survives is the live fragment: constructors, lambdas, matches and calls over live binders. The runtime then needs no garbage collector: a value has one owner, so a match frees its scrutinee as it opens it, arrays update in place, and a forked task carries no lock. Where a `+` licensed reuse the compiler places a counted share or a borrow, and nowhere else: reuse was derived from the kind of a type, never proved by a term, so the runtime never runs a copy the source did not spell.

6 The Mechanization

`bend2/bend.lean` mechanizes the core in Lean 4 [22]: about twenty thousand lines, no `sorry`, no axiom declarations. A de Bruijn specification mirrors `bend.ts` section by section; the proofs are confluence (`church_rosser_holds`), subject reduction for weak reduction (`subject_reduction_holds`) and, at the live demand, progress (`progress_holds`), weak normalization (`normalization_holds`) and consistency (`consistency_holds`).

The model covers the core of Section 2.4, with the `+` binder, kinds, Data and the meet, but it does not yet fully match the shipped checker: the file lists where `bend.ts` and the model differ, and we are resyncing the two. What the file proves stands: a calculus with Type : Type and negative datatypes, consistent and normalizing, guarded by affinity alone. Normalization and consistency are proven with recursive definitions included, by a Dershowitz–Manna multiset measure [11] over pending references. The dead boundary is a witness, not a caveat: in a well-formed book with a negative type, Curry's self-application term checks *dead* at an empty family and, by `consistency_holds`, never live.

7 Discussion

What affinity takes away. Contraction on closures. The standard `map` is ill-typed: `f` is applied once per element, so it would need `+`, and no function type is Data. Code is free, so a top-level definition may be called any number of times, and maps over a named function cover the common cases; the closure-heavy style of Haskell does not transfer. Bend accepts this on purpose: the runtime wants the same restriction [25].

What it keeps. On the program side the baseline is C: first-order data, machine words, arrays updated in place, code called by name. That fragment passes through affinity untouched. On the proof side the reach is larger: statements are erased and cost nothing, and live proofs draw their reuse from Data, which is where induction lives. A universe hierarchy could be added later; but Bend needs affinity anyway, and Type : Type buys impredicative encodings and type-computing definitions no predicative hierarchy accepts.

Related work. Linear and dependent types meet in LLF [7] and in [18]; the quantitative line [3, 4, 20] and the graded line [1, 21] are closest in mechanism, and all keep a universe hierarchy. Recent linear dependent theories still add universe levels to avoid Girard's paradox [13], or reach an impredicative universe through a model rather than Type : Type [24]. To our knowledge none uses the absence of contraction *for* consistency. Among mechanized metatheories of practical kernels [2, 5, 23], ours proves normalization rather than assuming it.

Limitations. The consistency result is syntactic, relative to Lean's own foundation, with no semantic model. The mechanization lags the shipped checker (Section 6). Equality is intensional, with no extensionality principle. And the theorems are about the calculus, not the code.

BendTT buys consistency with affinity instead of a universe hierarchy, earns reuse from the kind of a type instead of a proof, checks recursion with a descent rule an auditor can read in an afternoon, and erases everything that does not run. Nothing in it is difficult, and that is the point.

References

[1] Andreas Abel, Nils Anders Danielsson, and Oskar Eriksson. 2023. A Graded Modal Dependent Type Theory with a Universe and Erasure, Formalized. *Proceedings of the ACM on Programming Languages* 7, ICFP (2023), 220:1–220:35.

[2] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2018. Decidability of Conversion for Type Theory in Type Theory. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 23:1–23:29.

[3] Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2018. 56–65.

[4] Edwin Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming (ECOOP) (LIPIcs)*, 2021. 9:1–9:26.

[5] Mario Carneiro. 2024. Lean4Lean: Towards a Formalized Metatheory for the Lean Theorem Prover. Retrieved from https://arxiv.org/abs/2403.14064

[6] Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. 2014. Combining Proofs and Programs in a Dependently Typed Language. In *Proceedings of the 41st ACM Symposium on Principles of Programming Languages (POPL)*, 2014. 33–45.

[7] Iliano Cervesato and Frank Pfenning. 2002. A Linear Logical Framework. *Information and Computation* 179, 1 (2002), 19–75.

[8] Robert L. Constable and Scott F. Smith. 1987. Partial Objects in Constructive Type Theory. In *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science (LICS)*, 1987. 183–193.

[9] Thierry Coquand. 1991. An Algorithm for Testing Conversion in Type Theory. *Logical Frameworks*, 255–279.

[10] Haskell B. Curry. 1942. The Inconsistency of Certain Formal Logics. *Journal of Symbolic Logic* 7, 3 (1942), 115–117.

[11] Nachum Dershowitz and Zohar Manna. 1979. Proving Termination with Multiset Orderings. *Communications of the ACM* 22, 8 (1979), 465–476.

[12] Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *ACM Computing Surveys* 54, 5 (2021), 98:1–98:38.

[13] Qiancheng Fu and Hongwei Xi. 2023. A Two-Level Linear Dependent Type Theory. Retrieved from https://arxiv.org/abs/2309.08673

[14] Eduardo Giménez. 1994. Codifying Guarded Definitions with Recursive Schemes. In *Types for Proofs and Programs (TYPES) (LNCS)*, 1994. Springer, 39–59.

[15] Jean-Yves Girard. 1972. Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur. Doctoral dissertation.

[16] V. N. Grišin. 1982. Predicate and Set-Theoretic Calculi Based on Logic Without Contractions. *Mathematics of the USSR–Izvestiya* 18, 1 (1982), 41–59.

[17] Antonius J. C. Hurkens. 1995. A Simplification of Girard's Paradox. In *Typed Lambda Calculi and Applications (TLCA) (LNCS)*, 1995. Springer, 266–278.

[18] Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. 2015. Integrating Linear and Dependent Types. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*, 2015. 17–30.

[19] Luc Maranget. 2008. Compiling Pattern Matching to Good Decision Trees. In *Proceedings of the ACM SIGPLAN Workshop on ML*, 2008. 35–46.

[20] Conor McBride. 2016. I Got Plenty o' Nuttin'. *A List of Successes That Can Change the World 9600*, 207–233.

[21] Benjamin Moon, Harley Eades III, and Dominic Orchard. 2021. Graded Modal Dependent Type Theory. In *European Symposium on Programming (ESOP) (LNCS)*, 2021. Springer, 462–490.

[22] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction (CADE 28) (LNCS)*, 2021. Springer, 625–635.

[23] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 2020. Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 8:1–8:28.

[24] Sam Speight and Niels van der Weide. 2026. Impredicativity in Linear Dependent Type Theory. Retrieved from https://arxiv.org/abs/2602.08846

[25] Victor Taelin. 2026. *BendRT: A Parallel Runtime for CPUs and GPUs.*

[26] Kazushige Terui. 2004. Light Affine Set Theory: A Naive Set Theory of Polynomial Time. *Studia Logica* 77, (2004), 9–40.