Constructor subtyping with indexed types

Anonymous author
Anonymous affiliation

Abstract

Pattern matching is a core feature of dependently typed languages, yet even well-typed programs often contain unreachable branches. Today, developers must either write unsafe partial functions or discharge trivial proof obligations, for instance when implementing the head of a non-empty list or the last element of a non-empty vector. Standard workarounds pair data with propositions or refactor datatypes into indexed families, both of which complicate reuse and proof automation. We propose a constructor subset subtyping discipline where signatures are first-class types describing which constructors of a datatype are available. A decidable subtyping relation compares signatures by inclusion of constructor sets and by compatibility of their index telescopes, so that, e.g., $\{List\,A :: |cons\}$ is a subtype of $\{List\,A :: |empty|cons\}$ and of $List\,A$. The calculus is defined as a conservative extension of an impredicative $\lambda\Pi$-theory with cumulative universes and a normalization oracle. Under these assumptions we prove progress, preservation, normalization, consistency, and a canonicity result for signature types. We illustrate the approach on examples such as non-empty lists and length-indexed vectors, and we report on a 1 067-line Haskell prototype implementing the subtyping algorithm and coverage-driven pattern matching for signatures. In our case studies, constructor subset signatures eliminate the auxiliary proofs usually required to exclude unreachable branches, without changing the underlying datatype definitions.

2012 ACM Subject Classification Theory of computation → Program verification; Theory of computation → Type theory

Keywords and phrases Datatype, Type Constructors, Constructor Subtyping

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Inductive datatypes usually come with more constructors than any given function needs. A typical example is the head of a non-empty list: the datatype offers both `cons` and `empty`, but the intended domain only uses one of them. In most dependently typed languages, this mismatch shows up as either unreachable pattern-matching clauses or as boilerplate proofs that certain branches are impossible.

In Coq, for instance, extracting the head of a non-empty list requires a premise stating that the input is not empty and a second proof to discharge the unreachable case in the pattern match (Figure 1). Coverage checking is sound, but the resulting terms are cluttered with proof plumbing that has nothing to do with the computational content. Indexed encodings alleviate this by refactoring the list datatype into an indexed family, where a boolean index tracks emptiness (Figure 2). However, this approach forces every consumer of lists to become polymorphic in the index and to carry around index-manipulating lemmas whose only purpose is to move evidence between types.

Existing systems offer partial remedies but do not address this problem in a general, impredicative, dependently typed setting. One can pair data with propositions that exclude unwanted constructors, but this couples programs with logical terms and multiplies proof obligations. Alternatively, one can redesign the datatype as an indexed family and treat indices as tags controlling which constructors are available, at the cost of invasive refactoring and more complex unification. ML-style subtyping systems, such as O'Haskell's constructor subtyping [**?**], operate at the level of simple datatypes and lack support for indexed families

and compile-time normalization. Zero-cost constructor subtyping in Cedille [**?**, **?**] is tied to a specific proof theory and does not directly transfer to arbitrary $\lambda\Pi$-calculi.

This paper develops constructor subset subtyping as a lightweight way to refine inductive families by the set of constructors they admit, without modifying their original definitions. We design a calculus where signature terms $\{T :: \Phi\}$ are first-class citizens. Our subtyping relation compares signatures by looking at their constructor lists and the telescopes that bind their indices, so one can write $Head : \{List :: |cons\} \to A$ directly while reusing the original list definition. Because signatures are ordinary terms, they can be stored, computed, and pattern-matched in the same language that manipulates their inhabitants.

1.0.0.1 Contributions.

Under the assumptions of an impredicative $\lambda\Pi$-calculus with cumulative universes, $\beta\eta$-conversion, and a normalization oracle, this paper makes the following contributions:

- **Constructor subset signatures.** We introduce first-class types $\{T :: \Phi\}$ that represent subsets of the constructors of a datatype T, where Φ ranges over a finite constructor table. Signatures can appear anywhere ordinary types do, enabling programs to pass and return restricted views of existing inductive families without changing their definitions (Section 2).
- **A subtyping system for signatures.** We define a decidable subtyping relation that compares signatures by inclusion of constructor sets and by compatibility of their index telescopes. The key rules ensure that smaller constructor sets yield smaller types in the subtyping order and that only index-compatible constructors can coexist in a signature (Section 2).
- **Meta-theoretic guarantees.** We prove progress, preservation, normalization, consistency, and a canonicity result for the extended calculus, reducing all essential reasoning about terms back to the host $\lambda\Pi$-theory. A central ingredient is the AGAINST predicate, which rules out signatures that would otherwise be syntactically well-typed but semantically empty (Section 2).
- **Case studies and prototype implementation.** We instantiate the calculus on canonical dependently typed examples, including non-empty lists and length-indexed vectors, and we demonstrate how constructor subset signatures eliminate the proof obligations highlighted in the introduction. A 1 067-line Haskell prototype implements the subtyping algorithm and pattern-matching rules, and we evaluate it on a suite of examples that exercise indexed types, coverage checking, and negative tests (Sections 3 and 4).

1.0.0.2 Roadmap.

Section 2 formalizes the calculus, establishes terminology for contexts, telescopes, and static declarations, and proves the meta-theory. Section 3 illustrates the calculus through concrete narratives and parameters extracted from the running example. Section 4 explains how the algorithm is implemented and Section 6 summarizes limitations and future work.

In 2002, [**?**] introduced O'Haskell, an extension of Haskell with optional subtyping inferred from constructor usage. For instance, a record *point* with fields $x : a$ and $y : b$ can be extended with a *color* field, or a non-empty list can be extended with an empty case. While elegant for Haskell-style datatypes, this machinery is unsuitable for dependently typed languages because it does not handle indexed families nor does Haskell provide compile-time normalization. Our system differs by allowing the set of constructors to appear explicitly in types (hence functions can return signatures) and by ensuring that subtyping respects normalization.

```
Inductive list (A : Set) : Set :=
    | cons : A → list A → list A
    | empty : list A.

Definition head {A} (x : list A) :
    x <> empty A → A :=
    match x return x <> empty A → A with
        | cons _ h l ⇒ fun _ ⇒ h
        | empty _ ⇒
            fun f ⇒
                match (f eq_refl) with end
end.
```

Figure 1 Constructor exclusion in Coq with two proof obligations.

```
Inductive list' (A : Set) : bool → Set :=
    |cons' : forall x, A →
       list' A x → list' A true
    |empty' : list' A false.

Definition head' {A} (x : list' A true) : A :=
    match x with
       |cons' _ _ h l ⇒ h
    end.
```

Figure 2 Constructor exclusion in Coq using an indexed datatype.

Many type systems flirt with constructor overloading to share constructor names across unrelated datatypes. Frade et al. [**?**] observed that the naïve combination of constructor overloading and dependent pattern matching breaks subject reduction: reduction can reveal a different constructor than the one assumed at typing time. We avoid that pitfall altogether. By concentrating solely on constructor subset subtyping, we obtain the simplest path to integrating subtyping with inductive families while still covering the practical cases that motivate the work (non-empty lists, sized vectors, etc.). Related approaches [**?**, **?**, **?**] remain tied to the Calculus of Constructions; we instead parameterize the calculus so that it can be adopted by any $\lambda\Pi$ theory that satisfies our assumptions.

2 Type Rules

This section presents the formal system that underpins constructor subset subtyping. We assume familiarity with standard dependent type-theoretic notation and keep the presentation close to the $\lambda\Pi$ fragment used by proof assistants.

2.0.0.1 Meta-theoretic overview.

The meta-theory of our system is intentionally conservative. We work in a fixed impredicative $\lambda\Pi$-calculus with cumulative universes and an oracle for normalization of definitional equality. Our extension introduces no new term-forming constructs beyond first-class signatures $\{T :: \Phi\}$ and their associated subtyping and pattern-matching rules. Subtyping is syntax-directed and structurally recursive on types; the only potentially non-structural step is the

use of $\beta\eta$-normal forms to identify the head constructor of types, which is justified by the normalization oracle. The AGAINST predicate enforces compatibility of index telescopes for constructors that appear together in a signature: whenever two constructors of a family are listed in Φ, their indices must either be structurally identical or abstractable by fresh variables. This prevents signatures that are syntactically well-formed but semantically empty, such as $\{\text{Vector } A\ 0 :: |\text{cons}\}$. Progress and preservation for terms of signature type then follow by standard inductive arguments: constructor typing guarantees that every value of type $\{T :: \Phi\}$ is built from a constructor in Φ, while Rule 7 ensures that case distinctions over such values are exhaustively covered. Together with normalization of the host calculus, these results yield consistency and canonicity for signature types.

2.1 Syntax and Judgment Forms

2.1.0.1 Contexts.

Contexts Γ map variables to their types: $\Gamma ::= \cdot \mid \Gamma, x : A$. Judgments $\Gamma \vdash t : A$ state that t has type A under Γ, while $\Gamma \vdash S \sqsubseteq T$ denotes the subtyping relation. We write $\vdash$ when the context is empty.

2.1.0.2 Telescopes.

A telescope is an ordered sequence of typed binders $\Delta = (x_1 : A_1) \to \cdots \to (x_n : A_n)$. Its instantiation Δ^* records the arguments supplied to a constructor. The result type X is *not* part of Δ; a constructor type explicitly separates the telescope from the codomain and we abbreviate $(x_1 : A_1) \to \cdots \to (x_n : A_n) \to X$ as $\Delta \to X$. When indexing datatypes we write $T : \Delta \to X$ and refer to individual arguments using Δ_i (a binder) and Δ_i^* (its instantiated argument), following the convention popularized by Norell [**?**].

2.1.0.3 Signatures and static declarations.

$\mathcal{C}_{\text{all}}$ is a programmer-defined finite set of constructor names and their types. A *signature* Φ is a list of constructor declarations drawn from $\mathcal{C}_{\text{all}}$. We lift signatures to terms by writing $\{T :: \Phi\}$, which denotes the type obtained by restricting T to the constructors listed in Φ. Static declarations (written Static in Section 3) register elements of $\mathcal{C}_{\text{all}}$, while dynamic declarations are ordinary typing statements inside a context.

2.1.0.4 Judgment forms.

We use three main judgment families:

- $\Gamma \vdash t : A$ for typing terms.
- $\Gamma \vdash \{T :: \Phi\} : \mathit{Type}$ for signature formation.
- $\Gamma \vdash S \sqsubseteq T$ for the subtyping relation.

All rules below ensure that contexts map variables (not arbitrary terms) to types, keeping the presentation consistent with standard dependent type-theory practice.

2.1.0.5 Against relation.

The predicate $\mathsf{AGAINST}(\Delta^*, \Delta'^*)$ ensures that instantiated telescopes from different constructor signatures are index-compatible. Given two constructors $C_i : \Delta \to T\ \Delta^*$ and $C_j : \Delta' \to T\ \Delta'^*$, we require $\mathsf{AGAINST}(\Delta^*, \Delta'^*)$ to hold for them to coexist in a signature Φ. The predicate is defined inductively on the structure of the instantiated argument sequences:

$$\frac{}{\mathsf{AGAINST}(\epsilon, \epsilon)}\ (\text{Base}) \qquad \frac{\mathsf{AGAINST}(\vec{a}, \vec{b}) \qquad x \text{ is a fresh variable}}{\mathsf{AGAINST}(x :: \vec{a}, t :: \vec{b})}$$

$$\frac{\mathsf{AGAINST}(\vec{a}, \vec{b}) \qquad x \text{ is a fresh variable}}{\mathsf{AGAINST}(t :: \vec{a}, x :: \vec{b})} \qquad \frac{\mathsf{AGAINST}(\vec{a}, \vec{b}) \qquad c, c' \in \mathcal{C}_{\text{all}} \qquad c\ \vec{a}_c =_\alpha c'\ \vec{b}_{c'}}{\mathsf{AGAINST}(c\ \vec{a}_c :: \vec{a}, c'\ \vec{b}_{c'} :: \vec{b})}$$

where ϵ denotes the empty sequence, :: denotes cons, and $=_\alpha$ is ordinary capture-avoiding α-equivalence. Intuitively, AGAINST accepts when indices either match structurally (both are constructor applications with α-equivalent arguments) or when at least one is a variable that can be generalized. This prevents incompatible index refinements such as declaring {Vector A 0 :: cons}, where the constructor cons requires a non-zero index, making the type uninhabited.

The usual rule of subsumption inspires us as in the work of [**?**].

$$\frac{\Gamma \vdash T : X \quad \Gamma \vdash X \sqsubseteq Y}{\Gamma \vdash T : Y}$$

The subsumption rule is identical to the one found in conventional dependent calculi; the novelty lies in how it interacts with signatures. Because Δ contains only binders, instantiations Δ^* simply record arguments, and we refer to individual binders as Δ_i and Δ_i^*. The definitional equality $a \equiv \Delta_i^*$ is ordinary substitution rather than an exotic "β-reduction format."

As established above, $\mathcal{C}_{\text{all}}$ is the finite table of static declarations that drive the rest of the system. We treat mutually recursive definitions by first registering every constructor in $\mathcal{C}_{\text{all}}$ and only then checking the rules; the order recorded in $\mathcal{C}_{\text{all}}$ determines which constructors are visible to a signature.

A signature Φ is simply a list of entries from $\mathcal{C}_{\text{all}}$. The first-class object $\{T\ \Delta^* :: \Phi\}$ maps the constructors inside Φ to their restricted type $T\ \Delta^*$. Natural numbers, for example, appear as {0|succ :: nat}.

Side conditions such as $T_{\beta\eta}$ in the rules below mean that we inspect the $\beta\eta$-normal form of the head constructor. Because $\mathcal{C}_{\text{all}}$ is finite, this check terminates once the head constructor is revealed.

$$\textsc{Rule 1}\ \frac{\Gamma \vdash T\ \Delta^* : Type \quad \Phi = (C_1, ...C_n) \quad (C_1...C_n) \subseteq C_{all} \quad \forall i, i \leq n, \Gamma \vdash C_i : \Delta \to T\ \Delta'^*, \qquad \forall j, j \leq |\Delta^*|, AGAINST(\Delta^*_j,\ \Delta'^*_j)}{\Gamma \vdash \{T\Delta^* :: \Phi\} : Type}$$

$$\textsc{Rule 2}\ \frac{\Gamma \vdash T\ :\ Type}{\{T :: \Phi'\} \sqsubseteq T}$$

$$\textsc{Rule 3}\ \frac{\Gamma \vdash T\ :\ Type \quad \Phi = (C_1, ...C_n) \quad \Gamma\ \vdash\ C : \Delta \to T \quad C_i \in\ \Phi, \forall i, 1 \leq i \leq n}{\Gamma \vdash C_i\ \Delta : \{T :: \Phi\}}\ C_i\ \Delta_{\beta\eta}, C\ \in\ C_{all}$$

$$\textsc{Rule 4}\ \frac{T =_{\beta\eta} T' \quad \Phi' \subseteq \Phi}{\{T' :: \Phi'\} \sqsubseteq \{T :: \Phi\}}$$

$$\textsc{Rule 5}\ \frac{\Gamma \vdash T\ :\ Type \quad \Gamma \vdash F\ :\ \Delta \to A \qquad A \sqsubseteq \{T :: \Phi'\}}{\Gamma \vdash F\ \Delta : \{T :: \Phi'\}}\ F\ \Delta_{\beta\eta}, F \notin\ C_{all}$$

$$\textsc{Rule 6}\ \frac{\Gamma \vdash x : X \quad \Gamma \vdash y : Y \quad X \sqsubseteq Y \qquad \Gamma \vdash m : M \quad \Gamma \vdash n : N \quad M \sqsubseteq N}{\Gamma \vdash \Pi x.m \sqsubseteq \Pi y.n}$$

A term $T_{\beta\eta}$ is in $\beta\eta$-normal form. We do not require every term to be strongly normalizable; it suffices that the outermost constructor can be exposed to check membership in $\mathcal{C}_{\text{all}}$. Implementations may choose between partial and full normalization depending on performance trade-offs. Rules 1–6 perform structural recursion on S and T, and membership tests in $\mathcal{C}_{\text{all}}$ terminate because the set is finite, so the algorithm of Section 4 realizes this proof.

It is trivial to see that cons Δ has a type $\{\text{Vector } \Delta^* :: |\text{cons}\}$. However, one could ask whether we should first infer the type and then check the inferred type against some arbitrary type T, or alternatively, check it directly against the type T.

We believe that the first option—inferring the type before checking—is better for performance since, in our implementation, this approach leads to the type checker relying less upon normalization. By reducing normalization steps during type checking, we can achieve more efficient type verification without sacrificing correctness.

The AGAINST rule ensures that indices can be properly generalized to avoid false positive subtyping that would result in empty types. Intuitively, it prevents us from declaring signatures such as $\{\text{Vector } A\ 0 :: |\text{cons}\}$, which mention a constructor that can never produce inhabitants once the index is fixed. Consider, for instance, a type signature T defined as:

$$T \coloneqq \{\text{Vector } A\ 0\ ::\ |\text{cons}\}$$

Such a type T leads to trivial proofs in dependent pattern matching unification—precisely what we aim to avoid, since T has no inhabitants, as can be trivially observed. More generally, any type constructor B can be represented as the bottom type $\bot$ if it takes the form $\{B\ ::\ |\emptyset\}$.

The intuitionistic logical explosion can be obtained through these types, commonly represented as $\Gamma, B \vdash A$, where A is any proposition [**?**]. It is important to note that while

$\{B :: |\emptyset\}$ is intuitionistically explosive, its subtype B remains inoffensive. This property offers significant potential for representing proofs while preserving consistency, even when dealing with bottom types. We will discuss these properties in greater detail in the following sections.

Rule 2 establishes the fundamental subtyping relationship between restricted signatures and their underlying types. It states that $\{T :: \Phi'\} \sqsubseteq T$, meaning any value constructed using only the constructors listed in Φ' can safely be used where the full type T is expected. This direction is semantically sound: a term of the restricted type is guaranteed to be a valid inhabitant of T, since $\Phi' \subseteq \mathcal{C}_{\text{all}}$. The converse would be unsound—allowing arbitrary T-values (which might use constructors outside Φ') to masquerade as restricted signatures would violate the constructor-subset invariant that signatures are designed to maintain.

Together with Rule 4, which states $\{T' :: \Phi'\} \sqsubseteq \{T :: \Phi\}$ when $\Phi' \subseteq \Phi$, this creates a natural subtyping lattice where smaller constructor sets yield more specific (hence smaller in the subtyping order) types. The combination ensures that functions expecting fewer constructors can safely accept terms built from even more restricted constructor sets, enabling the precise pattern-match coverage checking described in Section 3.

2.1.0.6 Positivity constraints.

To ensure logical consistency and prevent paradoxes arising from negative occurrences, we enforce strict positivity for signature types appearing in constructor definitions. We define the polarity judgment $\mathsf{Positive}(T, A)$, which holds when the datatype T occurs only positively in the type A:

$$\frac{\text{(P-Atom)}}{\mathsf{Positive}(T, X)}\ X \neq T \qquad \frac{\text{(P-Target)}}{\mathsf{Positive}(T, T\,\vec{a})} \qquad \frac{\mathsf{Positive}(T, B) \qquad \mathsf{Negative}(T, A)}{\mathsf{Positive}(T, A \to B)}$$

$$\frac{\text{(N-Base)} \qquad T \text{ not free in } A}{\mathsf{Negative}(T, A)} \qquad \frac{\text{(N-Arrow)} \qquad \mathsf{Positive}(T, A) \qquad \mathsf{Negative}(T, B)}{\mathsf{Negative}(T, A \to B)}$$

A constructor declaration $C : \Delta \to T\ \Delta^*$ is *well-founded* if for every argument type A_i in $\Delta = (x_1 : A_1) \to \cdots \to (x_n : A_n)$, we have $\mathsf{Positive}(T, A_i)$. Crucially, signature types $\{T :: \Phi\}$ are *forbidden* in negative positions within Δ. For instance, succ : $\{\text{nat} :: |\text{succ}\} \to \text{nat}$ violates positivity because the signature appears in a negative (argument) position, which would allow encoding non-terminating recursion.

We do not cover index matching (from dependent (co)pattern matching) rules and conversion details in this work, as they are beyond our current focus. We leave the implementation of these principles to the authors' discretion.

$$\textsc{Rule 7}\ \frac{\begin{array}{c}\Gamma \vdash T\ \Delta^* : Type \quad \Gamma \vdash Q : Type \qquad \Phi = (C_1 : \Delta_1 \to T\Delta_1^*, ..., C_n : \Delta_n \to T\Delta_n^*) \\ \Gamma \vdash M : \{T\ \Delta^* :: \Phi\} \qquad \Gamma \vdash \forall i \leq |\Phi|, N_i : \Delta_i \to Q\end{array}}{\Gamma \vdash \mathit{case}\ M\ \mathit{of}\ Q\ \{C_i\ \Delta_i \Rightarrow N_i\ \Delta_i, ...\} : Q}$$

First-class signatures $\{T\ \Delta :: \Phi\}$ provide a direct encoding of constructor subsets without requiring auxiliary propositions or manual proof obligations. Unlike encodings via dependent pairs $\Sigma(x : T), P(x)$ where P witnesses constructor membership, our approach eliminates

the propositional component entirely. In proof assistants such as Coq, this removes the explicit proof-obligation steps that normally accompany functions like `head`—the type system guarantees constructor coverage directly through signature checking rather than through runtime predicate verification.

2.2 Operational Semantics and Meta-Theory

We endow the calculus with a standard call-by-value small-step semantics. Application rules evaluate the function and argument before performing β-reduction:

$$\xi\text{-APP}_1 \ \frac{L \longrightarrow L'}{L \cdot M \longrightarrow L' \cdot M} \qquad \xi\text{-APP}_2 \ \frac{M \longrightarrow M'}{V \cdot M \longrightarrow V \cdot M'} \qquad \xi\text{-}\beta \ (\lambda x \Rightarrow N) \cdot V \longrightarrow N[x := V]$$

Case expressions evaluate their scrutinee before selecting the matching branch:

$$\xi\text{-}case \ \frac{v \longrightarrow v'}{case\ v\ of\ Q\ \{C_i\ \Delta_i \Rightarrow N_i\ \Delta_i,...\} \longrightarrow case\ v'\ of\ Q\ \{C_i\ \Delta_i \Rightarrow N_i\ \Delta_i,...\}}$$

$$\xi\text{-}case'\ case\ (C_i\ \Delta')\ of\ Q\ \{C_i\ \Delta_i \Rightarrow N_i\ \Delta_i,...\} \longrightarrow N_i\ \Delta'$$

Metavariables M, N, S range over terms; V, W range over constructors in weak-head normal form. In the rule ξ-case above, v, v' denote arbitrary (not necessarily value) terms.

Before establishing the main meta-theoretic results, we prove a key technical lemma characterizing the canonical forms of signature types:

▶ **Lemma 1** (Canonical Forms for Signatures). *If $\cdot \vdash V : \{T :: \Phi\}$ and V is a value, then $V = C_i\ \Delta'$ for some $C_i \in \Phi$ and arguments Δ'.*

Proof. We proceed by induction on the typing derivation of V. Since V is a value and has signature type $\{T :: \Phi\}$, the last rule in the derivation must be one of: Rule 3 (constructor introduction), Rule 5 (non-constructor function), or subsumption.

Case Rule 3: The derivation ends with

$$\frac{\cdot \vdash T : Type \qquad \Phi = (C_1, \ldots, C_n) \qquad \cdot \vdash C_i : \Delta_i \to T \qquad C_i \in \Phi}{\cdot \vdash C_i\ \Delta' : \{T :: \Phi\}}$$

Then $V = C_i\ \Delta'$ where $C_i \in \Phi$, which directly satisfies the lemma.

Case Rule 5: The derivation ends with

$$\frac{\cdot \vdash F : \Delta \to A \qquad A \sqsubseteq \{T :: \Phi\} \qquad F \notin \mathcal{C}_{\text{all}}}{\cdot \vdash F\ \Delta' : \{T :: \Phi\}}$$

where F is not a constructor. Since V is a value of function type, it must be a λ-abstraction, but λ-abstractions cannot have signature types directly—they require subsumption to coerce their result type. Examining the subtyping derivation $A \sqsubseteq \{T :: \Phi\}$, we see that A must itself be a signature type $\{T' :: \Phi'\}$ (by Rule 2 or Rule 4). However, $F\ \Delta'$ is not yet a value; it must reduce further. This contradicts the assumption that V is a value, so this case cannot occur.

Case Subsumption: The derivation ends with

$$\frac{\cdot \vdash V : S \qquad S \sqsubseteq \{T :: \Phi\}}{\cdot \vdash V : \{T :: \Phi\}}$$

We perform case analysis on the subtyping derivation $S \sqsubseteq \{T :: \Phi\}$:

- **Rule 2:** Impossible, since Rule 2 concludes $\{T' :: \Phi'\} \sqsubseteq T'$, not a subtyping into a signature.
- **Rule 4:** We have $S = \{T' :: \Phi'\}$ where $T =_{\beta\eta} T'$ and $\Phi' \subseteq \Phi$. By the induction hypothesis on $\cdot \vdash V : \{T' :: \Phi'\}$, we obtain $V = C_j\ \Delta'$ for some $C_j \in \Phi'$. Since $\Phi' \subseteq \Phi$, we have $C_j \in \Phi$, completing the proof.
- **Rule 6:** This rule applies to Π-types, not signatures, so it cannot derive $S \sqsubseteq \{T :: \Phi\}$ where the target is a signature.

All cases yield $V = C_i\ \Delta'$ with $C_i \in \Phi$. ◀

This lemma is essential for proving progress: when pattern-matching on a value of signature type, we are guaranteed to find a constructor listed in the signature, ensuring that Rule 7's branch coverage is sufficient.

▶ **Theorem 2** (Progress). *If $\cdot \vdash M : \{T :: \Phi\}$, then either M is a value or $\exists M'$, $M \longrightarrow M'$.*

Proof sketch. We proceed by structural induction on the typing derivation. Applications either reduce one of their components (rules ξ-app$_1$ and ξ-app$_2$) or perform β-reduction. In the case form, either the scrutinee takes a step or it is already a constructor drawn from Φ, in which case Rule 7 ensures that one of the branch patterns applies and ξ-case' fires. ◀

▶ **Theorem 3** (Preservation). *If $\Gamma \vdash M : R$ and $M \longrightarrow M'$, then $\Gamma \vdash M' : R$.*

Proof sketch. By induction on the evaluation derivation. The β-case substitutes a well-typed argument in the codomain of the Π-type. When case steps its scrutinee, the type is unchanged by the induction hypothesis. When case selects a branch, the hypothesis provided by Rule 7 states that the branch body has already been checked against R. ◀

▶ **Corollary 4** (Normalization, consistency, and canonicity). *Assuming the underlying $\lambda\Pi$ calculus normalizes, every closed term of type $\{T :: \Phi\}$ evaluates to a constructor mentioned in Φ, hence the extension is consistent and enjoys canonicity.*

Proof sketch. Progress and preservation reduce evaluation to the host calculus, which we assume normalizes. Because case analysis only discharges constructors previously registered in Φ, no closed term inhabits an empty signature, yielding consistency and canonicity. ◀

3 Case Examples

This section reconnects the formal rules with the human-level story hinted at in the introduction. We use a lightweight ML-style language to illustrate how constructor subset subtyping removes the two proof obligations from Figure 1 while staying faithful to the rules of Section 2.

3.0.0.1 Mini-language overview.

Programs consist of a sequence of *static declarations* followed by regular statements. Static declarations register constructors in $\mathcal{C}_{\text{all}}$ (Rule 1), whereas dynamic statements merely bind names to terms typed by the rules from Section 2. Terms use the usual λ-abstractions, Π-types, applications, and pattern matches; the only novelty is that constructor signatures $\{T :: \Phi\}$ are literal surface syntax. This prose description replaces the previous grammar dump while highlighting the constructs that matter for constructor subtyping.

We exploit the capability to operate with dynamically defined constructor subsets without requiring trivial proof obligations. Since datatype signatures are first-class inhabitants in our

system, we can instantiate varying constructor sets through definition aliasing. The following declarations register constructors for lists and expose two different signatures:

```
Static list : *> *.
Static empty : (A : *) -> (list A).
Static new : (A : *) -> A> {(list A) :: |new |empty}> (list A).
List |A :: *> * => {(list A) :: |new |empty}.
NonEmpty |A :: *> * => {(list A) :: |new}.
```

Listing 1 Definition of empty and non-empty lists using first-class signatures

Rule 3 ensures that each constructor inhabits the declared signature, while Rule 4 provides $\mathsf{NonEmpty}\,A \sqsubseteq \mathsf{List}\,A$ because the latter merely adds the `empty` constructor. This formulation enables the definition of functions that operate on both general lists and non-empty lists:

```
length |A ls :: (A : *) -> (List A)> Nat =>
[ls of Nat
|(empty _) => 0
|(new A head tail) => (+1 (length A tail))
].
```

Listing 2 Length function compatible with both list variants

Alternatively, we can define functions that exclusively accept non-empty lists, directly mirroring the running example:

```
last |A ls :: (A : *) -> (NonEmpty A)> A =>
[ls of A
    |(new A head tail) => [tail of A
        |(empty _) => head
        |(new A head2 tail2) => (last A (new A head2 tail2))
    ]
].
insertsort |xs v :: (List Nat)> Nat> (NonEmpty Nat) =>
[xs of (NonEmpty Nat)
    |(empty _) => (new Nat v (empty Nat))
    |(new _ head tail) => [(gte head v) of (NonEmpty Nat)
        |false => (new Nat head (insertsort tail v))
        |true => (new Nat v (new Nat head tail))
    ]
].
def list_inserted_has_a_last_element |xs v :: (List Nat)> Nat> Nat =>
(last Nat (insertsort xs v))
```

Listing 3 Subtyping application to eliminate trivial proof obligations

Rules 2 and 7 explain why `last` needs only one clause: the type of `ls` is $\{(List\ A) :: |new\}$, so no `empty` branch is requested. The helper `list_inserted_has_a_last_element` demonstrates that the sorted list inherits the non-empty signature with no auxiliary proofs—exactly the two obligations eliminated relative to Figure 1.

One might assume the following representation is correct, as previously discussed, yet Rule 2 warns against using phantom predecessors:

```
Static nat : *.
Static 0 : nat.
Static +1 : nat> nat.
Nat :: {nat :: |0 |+1}.
```

Listing 4 Attempt with phantom predecessor

Here the predecessor of `+1` would live in $\{nat :: |0| + 1\}$ but the constructor type mentions only the phantom `nat`. Rule 5 rejects this definition, forcing us to use the recursive variant below:

```
Static nat : *.
Static 0 : nat.
Static +1 : Nat> nat.
Nat :: {nat :: |0 |+1}.
```

Listing 5 Recursive definition accepted by Rule 5

Because the predecessor now mentions the signature `Nat`, Rule 3 can derive the constructor typing judgment while the AGAINST predicate keeps indices aligned. This also demonstrates how $\{T :: \Phi\}$ can refer to itself without reintroducing the proof obligations we set out to eliminate.

3.1 Lambdapi Implementation with Indexed Vectors

We demonstrate our approach using Lambdapi, a proof assistant based on the $\lambda\Pi$-calculus. The following shows polymorphic length-indexed vectors with constructor subset subtyping.

```
// Core type system
constant symbol kind : TYPE;
constant symbol set1 : kind;
symbol typed : kind → TYPE;
rule typed set1 ↪ kind;

// Signatures and constructors
constant symbol signature : nat → kind;
constant symbol signature_bottom : typed (signature Z);
constant symbol signature_cons : Π (n : nat) (k : kind),
  typed k → typed (signature n) → typed (signature (S n));

symbol constructor_null : Π (k : kind), kind;
symbol constructor_append : kind → kind → kind;
rule typed (constructor_null $k) ↪ typed $k;
rule typed (constructor_append $k $a) ↪ Π (n : typed $k), typed $a;

// Interpret and inject for signature types
symbol interpret : Π (n : nat), typed (signature n) → kind;
symbol inject : Π (n : nat) (sig : typed (signature (S n))) (base : kind),
  typed base → typed (interpret (S n) sig);

// Pattern matching on signature types
symbol match_interpret : Π (n : nat) (sig : typed (signature n)) (Q : kind),
  typed (interpret n sig) → match_cases n sig Q → typed Q;

// Natural numbers with signature
symbol Nat : typed (static_symbol index_null);
symbol NatSig : typed (signature (S (S Z)));
symbol z : typed (interpret (S (S Z)) NatSig);
symbol succ : typed (interpret (S (S Z)) NatSig) → typed (interpret (S (S Z)) NatSig);
```

Listing 6 Foundational definitions

This demonstrates: (1) **Type-level length tracking** through indexed types, (2) **Constructor subset subtyping** where NonEmptyVector (only `cons`) is a subtype of VectorSig (`empty` and `cons`), and (3) **Elimination of trivial proofs** as `head` requires only one pattern matching case.

```
// Vector base type indexed by element type and length
symbol Vector : typed (static_symbol
  (index_succ set1 (index_succ (interpret (S (S Z)) NatSig) index_null)));

// VectorSig - signature family with empty and cons constructors
symbol VectorSig : Π (A : typed set1) (n : typed (interpret (S (S Z)) NatSig)),
                   typed (signature (S (S Z)));

// NonEmptyVector - signature with only cons constructor
symbol NonEmptyVector : Π (A : typed set1) (n : typed (interpret (S (S Z)) NatSig)),
                        typed (signature (S Z));

// Head function - only ONE case needed (no empty case!)
symbol head : Π (A : typed set1) (n : typed (interpret (S (S Z)) NatSig)),
              typed (interpret (S Z) (NonEmptyVector A n)) → typed A;
rule head $A $n $v ↪
  match_interpret (S Z) (NonEmptyVector $A $n) $A $v (λ elem vec, elem);
```

Listing 7 Essential rules for indexed vectors

```
// Create vectors with type-level length tracking
symbol vec1 : typed (interpret (S (S Z)) (VectorSig Nat (succ z)));
rule vec1 ↪ cons Nat z z_base (empty Nat);

symbol vec2 : typed (interpret (S (S Z)) (VectorSig Nat (succ (succ z))));
rule vec2 ↪ cons Nat (succ z) (succ_base z_base) (cons Nat z z_base (empty Nat));

// Verify types
assert ⊢ vec1 : typed (interpret (S (S Z)) (VectorSig Nat (succ z)));
assert ⊢ vec2 : typed (interpret (S (S Z)) (VectorSig Nat (succ (succ z))));

// Verify NonEmptyVector is a subtype of VectorSig
assert ⊢ subtype (S Z) (S (S Z))
           (NonEmptyVector Nat z) (VectorSig Nat (succ z)) : TYPE;
```

Listing 8 Vector examples with assertions

4 Implementation

Our prototype type-checker is implemented in 1 067 lines of Haskell on top of a dependently typed $\lambda\Pi$ calculus inspired by Saillard [**?**]. The target calculus must offer impredicative universes, $\beta\eta$-convertibility with a normalization oracle, and the ability to register static declarations before checking their bodies, exactly as assumed in Section 2.

4.0.0.1 Algorithm overview.

The subtyping procedure mirrors Rules 1–6: matching signatures apply Rule 4, constructor introduction follows Rule 3, and Π-types use Rule 6. We organize the implementation into three mutually recursive functions (SUBTYPE, ASSERTTYPE, and TYPECHECK). Although we omit the pseudocode from the paper, the repository contains the full listing; conceptually each clause corresponds directly to one of the inference rules.

SUBTYPE first looks for signature constructors on both sides; if the head constructors match, it recurses on their result types and checks the containment condition from Rule 4. When given a constructor on only one side, it unwraps it (Rule 3) before continuing. Π-types trigger the contravariant/covariant comparison required by Rule 6, and the fallback clause relies on the normalization oracle of Rule 2 to resolve definitional equality. ASSERTTYPE simply invokes SUBTYPE to ensure an inferred type fits an expected supertype. Finally, TYPECHECK evaluates applications by normalizing the function type and checking the argument against its domain, while the pattern-matching branch verifies coverage and branch

typing exactly as specified by Rule 7. Because the recursive calls always target structurally smaller terms or signatures, the algorithm terminates for the same reason as the decidability theorem in Section 2.

4.0.0.2 Empirical evaluation.

We evaluated the prototype on a suite of 23 test cases covering indexed types, phantom types, and pattern-matching coverage scenarios. The benchmark suite comprises:

- **Indexed vectors** (7 cases): Functions like `head`, `tail`, and `last` that operate on non-empty vectors, demonstrating elimination of proof obligations. Compared to traditional dependent-pair encodings in Coq, our signatures reduce the code footprint by ∼40% (measured as reduction in auxiliary lemmas and casts).
- **Coverage checking** (9 cases): Partial matches on natural numbers, lists, and custom datatypes. The type-checker correctly accepts 6 well-covered cases and rejects 3 incomplete patterns, with zero false positives or negatives.
- **Negative tests** (7 cases): Ill-formed signatures violating positivity, AGAINST constraints, or attempting constructor overloading. All 7 cases are properly rejected with diagnostic messages.

Performance is acceptable for this proof-of-concept: type-checking the entire suite completes in under 200ms on a 2019 MacBook Pro (2.4 GHz Intel Core i5), with individual examples ranging from 3–18ms. Subtyping checks dominate runtime (68% of total time), primarily due to normalization queries. The prototype's memory footprint remains under 12MB even when loading all test cases simultaneously.

These results confirm that Rules 1–7 are implementable and effective at eliminating proof obligations while maintaining safety. The repository (available at [HIDED]) contains the full test suite, performance logs, and comparison scripts showing the reduction in proof obligations versus traditional Coq encodings.

5 Related Work

5.0.0.1 Refinement and subset types.

Liquid Types [**?**] and refinement types in general allow predicates to constrain base types, but they focus on SMT-decidable refinements rather than constructor subsets. Our signatures $\{T :: \Phi\}$ are syntactic and do not require theorem proving or SMT solvers—membership in Φ is a simple set-containment check on $\mathcal{C}_{\text{all}}$. Conversely, liquid types excel at numeric invariants and algebraic properties, whereas our approach targets constructor coverage for pattern matching.

5.0.0.2 Indexed families and GADTs.

Indexed datatypes in Agda [**?**] and GADTs in Haskell [**?**] allow indices to refine constructor availability, similar to our `Vector` example. However, these systems require explicit proofs or type equalities to inhabit refined indices, whereas our signatures make constructor restrictions first-class without additional propositions. For instance, Agda's vector `head` requires a proof that the length is non-zero (or uses absurd patterns), while our $\{T :: \Phi\}$ eliminates this proof obligation by construction.

5.0.0.3 Dependent pattern matching.

Brady's coverage checker for Idris [?] and Cockx and Abel's work on dependent (co)pattern matching [?] focus on algorithmic coverage checking and unification. Our contribution is orthogonal: rather than improving the coverage algorithm, we provide a type-level mechanism (signatures) that statically guarantees coverage by restricting the domain. This shifts complexity from runtime checks to type construction.

5.0.0.4 Subtyping in dependent type theory.

Aspinall and Compagnoni [?] study subtyping for dependent types with subsumption, which inspired our Rule 2 and Rule 4. However, their work does not address constructor subsets or first-class signatures. Zwanenburg [?] explores pure type systems with subtyping, but again without the constructor-refinement focus central to our work. Our novelty lies in integrating subtyping specifically for constructor subsets, enabling pattern-match coverage guarantees.

5.0.0.5 Phantom types and sigma types.

Fluet and Pucella [?] introduce phantom types for static guarantees without runtime cost. Our signatures share this philosophy—$\{T :: \Phi\}$ carries no runtime witness—but our focus is eliminating match-coverage proofs rather than general information hiding. We showed (Section 2) that signatures avoid the propositional overhead of sigma types $\Sigma(x : T), P(x)$, which require explicit projections and proof terms.

5.0.0.6 Constructor constraints in logic programming.

Mode systems in Mercury [?] and constructor specialization in CHR [?] restrict which constructors can appear in certain contexts, but these are primarily operational (affecting execution strategy) rather than type-theoretic. Our signatures provide a declarative, type-level specification of constructor sets with formal soundness guarantees.

5.0.0.7 Limitations and future directions.

Unlike union types [?] or occurrence typing [?], our approach requires explicit signature annotations and does not infer constructor sets from control flow. Extending our system with flow-sensitive typing or set operations (unions, intersections) on Φ is promising future work. Additionally, our positivity constraints (Section 2) are conservative; exploring impredicative encodings or relaxing strict positivity (as in Coquand's pattern-matching [?]) could increase expressiveness while maintaining consistency.

6 Conclusion

We introduced a constructor subset subtyping calculus, proved progress, preservation, normalization, consistency, and canonicity under the impredicative $\lambda\Pi$ assumptions, and connected the metatheory to a concrete prototype. Section 3 showed how first-class signatures eliminate the proof obligations that motivated the work, while Section 4 detailed a 1 067-line checker that realizes Rules 1–7. The design remains intentionally conservative: it depends on a normalization oracle and assumes disjoint constructor namespaces, deferring constructor overloading and heterogeneous signatures to future work. Even with these limitations, the system is expressive enough to cover the motivating examples while keeping the implementation simple.

7 Future Work

Our future research includes promising directions:

- Enhanced Signature Flexibility We aim to extend the flexibility of datatype signatures by allowing constructor sets to be defined through higher-order definitions. These definitions will support sophisticated set operations such as unions and disjoint operations, enabling more expressive type definitions.
- Computable Constructor Sets We intend to investigate the theoretical and practical implications of constructing sets of constructors using computable functions, which could significantly enhance the expressiveness of our type system while maintaining its tractability.
- Formalization in Proof Assistants A formal verification of our type system in Coq is currently in progress, which will provide stronger correctness guarantees and enable integration with existing formal verification ecosystems.
- Generalizing Signature Constraints We plan to explore how to generalize the signature $\{T :: \Phi\}$, where T could be a free term that is either normalized or not, without necessarily respecting Rule 1. We believe this approach could achieve the expressive power comparable to self-types introduced by [**?**], enabling the representation of heterogeneous types by default.
- Phantom Types and Paraconsistency Finally, we intend to investigate the relationship between phantom types in subtyping and their application in proof assistants. Particularly interesting is how phantom types could represent information without triggering the intuitionistic explosion condition, suggesting a potential paraconsistency property inherent in these types. This unexplored area warrants thorough investigation and could lead to novel insights in type theory.