Constructor subtyping with indexed types

Anonymous author
Anonymous affiliation

Abstract

Pattern matching is a powerful feature of functional and dependently typed languages, yet programmers often encounter unreachable clauses. This either leads to unsafe partial functions or, in safe settings, to proof obligations that require tedious resolution of trivial cases.

The problem is particularly acute when programs operate on subsets of an inductive datatype. Standard encodings pair data with propositions or refactor datatypes into indexed families, but both approaches complicate composition and proof automation. We present a subtyping discipline for constructor subsets, where first-class signatures describe which constructors of a family are available. Our prototype shows that these signatures can eliminate many trivial proof obligations while preserving type safety.

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

Subtyping is a fundamental concept across programming languages that enables the specialization of type definitions through hierarchical relationships between supertypes and subtypes. At a high level, a subtype describes a "smaller" collection of values than its supertype. While term-level subtyping is extensively implemented in many languages, subtyping between constructors of inductive types remains relatively unexplored, especially in the presence of dependent types.

A recurring difficulty arises when an inductive datatype is only partially available in a given context. For example, a function may be intended to consume lists that are known to be non-empty, or vectors whose length is positive. In most dependently typed languages the programmer faces a choice: either write an unsafe partial function, or thread propositions through the program and discharge proof obligations stating that certain constructors are impossible. The latter approach leads to unreachable clauses in pattern matches and to proofs that are often routine but cumbersome.

One common technique is to maintain the original datatype together with a proposition that rules out some constructors. Another approach, illustrated in Figures 1 and 2, uses indexed datatypes to restrict which constructors may appear. Both techniques have well-known drawbacks. The first ties ordinary programs to logical predicates, complicating function composition and proof reuse. The second introduces additional indices and can interact badly with proof irrelevance and unification; it may also hinder the detection of unreachable branches when constructor information flows through computations.

Constructor subtyping has been explored in other settings. In O'Haskell [10], records and algebraic datatypes can be equipped with an optional subtyping relation inferred from the usage of constructors and fields. For example, a non-empty list type can be extended with an empty constructor to obtain a standard list type. More recent work on zero-cost constructor subtyping in Cedille [7, 6], on order-sorted inductive types and overloading [3, 2], and on algebraic subtyping for extensible records [8] shows how rich subtyping disciplines can be added to strongly typed systems. However, these systems are tailored to specific

```
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 a proof obligation.

core calculi or to non-dependent settings, and do not directly address the combination of constructor subtyping with indexed families in a general dependently typed setting.

In this paper we focus on a simple but expressive fragment of constructor subtyping aimed at eliminating trivial proof obligations. Rather than allowing arbitrary constructor overloading, we work with *constructor subsets*: first-class types of the form $\{T :: \Phi\}$, where T is an inductive family and Φ is a finite list of its constructors. Intuitively, $\{T :: \Phi\}$ denotes the values of T that are built only from constructors in Φ. We restrict attention to subtyping relations induced by inclusion of constructor sets and by compatibility of the indices of T.

Our contributions are as follows.

- We define a subtyping discipline for constructor subsets as a conservative extension of a $\lambda\Pi$-style dependent type theory with inductive families. The system includes typing and subtyping rules for constructor signatures and a dedicated elimination rule for pattern matching on signature types.
- We show, through a series of examples, that constructor subsets capture common programming idioms such as non-empty lists and simple invariants on inductive families, while avoiding the proof obligations that arise when these invariants are expressed with explicit propositions.
- We describe a small, direct prototype type checker in Haskell that implements the subtyping rules and coverage-driven pattern matching for constructor signatures, together with a compact Lambdapi formalization of the same system. The size and structure of this formalization support our claim that the calculus can be adapted to other impredicative dependent type systems with little overhead.

Structure of the paper.

Section 2 introduces the core type system and subtyping rules. Section 3 presents case studies illustrating how constructor subsets simplify programming with indexed families. Section 4 sketches the implementation and discusses practical considerations, and Section B concludes with directions for future work.

Many type systems are inspired by constructor subtyping to address constructor overloading, allowing the same constructor name to be shared by different datatypes, even in pattern matching. However, its naive use can lead to problems with subject reduction, as observed by Frade [5]. Subject reduction is the property that during the normalization of a term its type is preserved; overloading constructors without care may violate this property.

```
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.

For the sake of simplicity, in this work we do not consider overloading between distinct datatypes. Instead we restrict subtyping to relations induced by subsets of constructors of a single inductive family. We believe this fragment is already expressive enough to avoid the trivial proof obligations discussed above, while keeping the meta-theory and implementation comparatively simple.

2 Type Rules

In this section, we introduce a simplified type system, enhanced with first-class signature subtyping support. This exposition assumes a basic familiarity with the concepts of type theory, especially dependent type theory. We use a standard typing judgment $\Gamma \vdash t : A$ for terms and a subtyping judgment $\Gamma \vdash A \sqsubseteq B$ between types. Contexts Γ map variables to types, and $Type$ denotes the universe of small types.

The subtyping relation is inspired by the subsumption rule of Aspinall and Compagnoni [1]:

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash A \sqsubseteq B}{\Gamma \vdash t : B}$$

We now fix some notation for telescopes and constructor environments. A *telescope* Δ is a sequence of typed variables

$$\Delta = (a_1 : A_1) \dots (a_n : A_n).$$

Given such a telescope and a type family T, we write $T\,\Delta$ for the application $T\,a_1\,\dots\,a_n$. When a telescope is used only to record the indices of a datatype we write Δ^*; in examples we use $Vector\,A\,n$ with $\Delta = (A : Type)(n : Nat)$ and $\Delta^* = (A, n)$ [11].

To refer to individual arguments in a telescope we use subscripts: if $\Delta = (a_1 : A_1) \dots (a_n : A_n)$ then Δ_i denotes $(a_i : A_i)$ and Δ_i^* denotes the corresponding index, for $1 \le i \le n$. For a term T we use the side condition $T_{\beta\eta}$ to indicate that we inspect the $\beta\eta$-normal form of T.

We write $\mathcal{C}_{\text{all}}$ for the finite set of static constructor declarations provided by the programmer. A static definition is a name of the constructor, notice that we do not care how constructors are labeled, we only need to know when they are equal, one may use unique names or hashes for that. The order of static definitions is stored in $\mathcal{C}_{\text{all}}$; in particular, mutually recursive definitions can be handled by first registering every constructor in $\mathcal{C}_{\text{all}}$ and only then checking the rules below. A *signature* Φ is a finite list of constructor names drawn from $\mathcal{C}_{\text{all}}$, and a type of the form $\{T\,\Delta^* :: \Phi\}$ maps each constructor in Φ to the restricted

type $T\,\Delta^*$. We use a side predicate $\mathsf{AGAINST}(\Delta^*, \Delta'^*)$, defined later in this section, to ensure that indices in different constructors remain compatible. Because $\mathcal{C}_{\text{all}}$ is finite, the side conditions that test membership in $\mathcal{C}_{\text{all}}$ are decidable.

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

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

$$\textsc{Rule 3}\ \frac{\begin{array}{c}\Gamma \vdash T\ :\ Type \quad \Phi = (C_1, ...C_n) \\ \Gamma\ \vdash\ C : \Delta \to T \quad C_i \in\ \Phi, \forall i, 1 \leq i \leq n\end{array}}{\Gamma \vdash C_i\,\Delta : \{T :: \Phi\}}\ C_i\,\Delta_{\beta\eta}, C\ \in\ \mathcal{C}_{\text{ALL}}$$

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

$$\textsc{Rule 5}\ \frac{\Gamma \vdash T\ :\ Type \quad \Gamma \vdash F\ :\ \Delta \to A \qquad \Gamma \vdash A \sqsubseteq \{T :: \Phi'\}}{\Gamma \vdash F\,\Delta : \{T :: \Phi'\}}\ F\,\Delta_{\beta\eta}, F \notin\ \mathcal{C}_{\text{ALL}}$$

$$\textsc{Rule 6}\ \frac{\Gamma \vdash A' \sqsubseteq A \qquad \Gamma, x : A' \vdash B \sqsubseteq B'}{\Gamma \vdash (x : A) \to B \sqsubseteq (x : A') \to B'}$$

A term annotated as $T_{\beta\eta}$ is considered in $\beta\eta$-normal form. In practice we only normalize enough to inspect the outermost constructor when applying Rules 3 and 5; there is no global requirement that all terms be strongly normalizing. The choice of how aggressively to normalize is left to the implementation and presents the usual trade-off between completeness of simplification and performance.

It is trivial to see that cons Δ has type $\{\text{Vector}\ \Delta^* :: |\text{cons}\}$. One may either infer this type and then check it against some expected type T, or attempt to check cons Δ directly against T. In our prototype we follow the former strategyfirst infer, then checkbecause it tends to reduce the number of normalisation steps needed during type checking.

The AGAINST rule ensures that indices can be properly generalized to avoid false positive subtyping that would result in empty types. 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 unificationprecisely 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 [9]. 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.

Now we can introduce the $AGAINST$ rules that try to generalize indexed datatypes between constructors. You can see $AGAINST$ has a predicate, so indexed values of different constructs have to respect these rules. The equality $=_\alpha$ simply means the alpha-equivalence relation.

$$\frac{}{AGAINST(\emptyset, \emptyset)} \qquad \frac{AGAINST(\Delta, \Delta') \quad v\ is\ Var}{AGAINST(v\ ...\ \Delta, c\ ...\ \Delta')}$$

$$\frac{AGAINST(\Delta, \Delta') \quad (c, c') \subseteq \mathcal{C}_{\text{all}} \qquad c\ \Delta^c =_\alpha\ c'\Delta^{c'}}{AGAINST((c\ \Delta^c)\ ...\ \Delta, (c'\ \Delta^{c'})\ ...\ \Delta')}$$

Rule 2 allows us to forget signature information: any value of type $\{T :: \Phi'\}$ can be viewed as a value of the underlying type T. In this sense constructor subsets behave like *phantom types* [4]: the extra information carried by the signature is present at type-checking time but is not reflected in the runtime representation of values.

Though phantom types are not widely known across programming languagesbeing primarily used in Haskellthey possess important properties for representing types that carry no information during runtime; that is, they are purely erased types [4]. In our system there is no dedicated elimination rule for pure phantom signatures, and Rule 7 only permits case analysis on signatures that still expose constructors.

A constructor C must adhere to certain restrictions in its definition. For example, the signature (i.e., a type $\{T\ \Delta^*\ ::\ \Phi\}$) cannot occur freely in C. This restriction prohibits constructor types such as succ : $\{\text{nat} :: |\text{succ} \mid 0\} \to \{\text{nat} :: |\text{succ} \mid 0\}$, which would lead to problematic self-reference, as succ would require itself in its own definition.

It is important to note that even $succ : nat \to nat$ constitutes an invalid definition in this system. This is because the predecessor is a phantom type; therefore, for C to satisfy the inductive criteria, a signature must occur freely only in the positive position in the Δ of succ.

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.

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

It's easy to see that the first-class $\{T\ \Delta :: \Phi\}$ corresponds to the types discussed above with type constructors specialization. The advantage is that eliminates any form of proposition holding these types. In proof assistants like Coq, this could be seen as the elimination of some proof obligation steps.

▶ **Theorem 1.** *Let T be an inductive family with constructor set C, and let $S \subseteq C$. Define a predicate $D : T \to Type$ such that for every constructor application $C'\,\Delta$,*

$$D(C'\,\Delta) \;\equiv\; \begin{cases} \top & \text{if } C' \in S, \\ \bot & \text{if } C' \notin S. \end{cases}$$

Then $\{T :: S\} \simeq \Sigma(x : T), D(x)$.

Proof sketch. We define mutually inverse functions between $\{T :: S\}$ and $\Sigma(x : T), D(x)$.

- Given $x : \{T :: S\}$, the underlying term of x is built only from constructors in S claimed by the Rule 7. Therefore $D(x)$ is provable (by construction of D), and we obtain

$$f(x) \;:=\; (x, d_x) : \Sigma(y : T), D(y)$$

  for some trivial witness $d_x : D(x)$.
- Conversely, given a pair $(y, d) : \Sigma(x : T), D(x)$, the proof $d : D(y)$ guarantees that y was built using only constructors from S. Thus y inhabits $\{T :: S\}$, and we define

$$g(y, d) \;:=\; y : \{T :: S\}.$$

By construction we have $f(g(y, d)) = (y, d)$ and $g(f(x)) = x$, since f simply re-attaches a canonical proof of $D(x)$ and g forgets it. Hence f and g witness the claimed isomorphism. ◀

$$\xi\text{-}\cdot\text{APP}_1\ \frac{L \longrightarrow L'}{L \cdot M \longrightarrow L' \cdot M} \qquad \xi\text{-}\cdot\text{APP}_2\ \frac{M \longrightarrow M'}{V \cdot M \longrightarrow V \cdot M'}$$

$$\xi\text{-}\beta\ \frac{}{(\lambda x \Rightarrow N) \cdot V \longrightarrow N[x := V]}$$

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

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

Metavariables M, N, S range over terms; V, W range over values (constructors in WHNF). In the rule ξ-case above, the symbols v, v' denote arbitrary terms (not values).

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

Proof sketch. We reason by structural induction on M and case analysis on its form.

- *Application $M = M_1 \cdot M_2$.* If either M_1 or M_2 is not a value, the induction hypothesis yields M_i' with $M_i \longrightarrow M_i'$, and we use the corresponding congruence rule ξ-app$_i$ to obtain a step for M. Otherwise both M_1 and M_2 are values. If M_1 is a lambda abstraction $\lambda x.N$, we can perform a β-reduction step. If M_1 is a constructor, then M is already a value (an applied constructor) and the conclusion holds.

- *Case expression* $M = \mathit{case}\ v\ \mathit{of}\ Q\ \{C_i\,\Delta_i \Rightarrow N_i\,\Delta_i, \ldots\}$. If v is not a value, the induction hypothesis gives v' with $v \longrightarrow v'$, and we use rule ξ-case. If v is a value, then by typing rule 7 we have $v : \{T' :: \Phi'\}$ and $Q = \{T :: \Phi\}$. There must be some pattern $C_i : \Delta_i$ that matches v: by Rule 3 we know that $C_i \in \Phi'$, and by Rule 7 we have a corresponding branch $C_i : \Delta_i \to T : \Delta^*$. In this situation we can apply rule ξ-case' and M takes a step.
- *Other forms.* For variables, constructors, and abstractions the canonical forms analysis from the typing rules shows that well-typed closed terms of signature type are either values or reduce by one of the cases above.

In all cases, a closed term of type $\{T :: \Phi\}$ is either a value or can take a reduction step. ◀

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

Proof sketch. We proceed by induction on the evaluation derivation, considering the last reduction rule used.

- *β-reduction.* If $M = (\lambda x.N) \cdot V$ and $M' = N[x := V]$, typing gives $\Gamma \vdash \lambda x.N : (x : A) \to B$ and $\Gamma \vdash V : A$. By the substitution lemma we obtain $\Gamma \vdash N[x := V] : B[x := V]$, which is the same type as M.
- *ξ-app$_1$ and ξ-app$_2$.* In these cases we reduce a proper subterm of M. The induction hypothesis states that the reduced subterm preserves its type; reapplying the application typing rule reconstructs a typing derivation for M' with the same type R.
- *ξ-case.* Here $M = \text{case}\ v\ \text{of}\ Q\ \{\ldots\}$ and $M' = \text{case}\ v'\ \text{of}\ Q\ \{\ldots\}$ with $v \longrightarrow v'$. By the induction hypothesis $\Gamma \vdash v' : \{T :: \Phi\}$, and Rule 7 then re-establishes $\Gamma \vdash M' : Q = R$.
- *ξ-case'.* In this case $M = \text{case}\ (C_i\ \Delta')\ \text{of}\ Q\ \{\ldots\}$ reduces to $M' = N_i\ \Delta'$. Typing rule 7 requires that for each branch we have $\Gamma \vdash N_i : \Delta_i \to Q$, and that $C_i : \Delta_i \to T\ \Delta^*$. Instantiating with the actual arguments Δ' shows that M' has type Q, the same type as M.

All other reduction rules follow the same pattern: either a proper subterm reduces and the induction hypothesis applies, or a local computation such as β-reduction is justified by the typing rules. In each case the result has the same type R as the original term. ◀

3 Case Examples

We introduce a minimal language and its implementation to demonstrate the subtyping system. The abstract syntax of terms is sketched in Figure 3; in the examples below we use a lightweight ML-style surface syntax.

$$\begin{aligned}\text{Term} ::= {} & \text{Var}(x) \mid \text{App}(f, a) \mid \text{Lam}(x, e) \mid \text{Pi}(x, A, B) \\ & \mid \text{Constr}(T, \overrightarrow{c}) \mid \text{Match}(e, T, \overrightarrow{p \to e'}) \mid \text{Notation}(e, T)\end{aligned}$$

Figure 3 Term syntax of the example language

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 bind names to terms typed by the rules from Section 2. Terms use the usual λ-abstractions, function types, applications, and pattern matches; the only novelty is that constructor signatures

$\{T :: \Phi\}$ appear explicitly in types. These informal descriptions suffice for the examples below; the full concrete syntax is implemented in the prototype but omitted here.

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 [9].

4 Implementation

Our prototype type checker is implemented in Haskell on top of a dependently typed $\lambda\Pi$ calculus modulo rewriting [12]. The implementation closely follows Rules 1–7 from Section 2: subtyping is a separate, decidable judgment used by the typing rules via subsumption, and coverage checking for pattern matching over signatures is driven by the constructor list Φ, so that functions expecting non-empty signatures are not forced to define unreachable branches. The current implementation is approximately 1 067 lines of code (including conversion and unification) and is accompanied by a Lambdapi formalization of the core constructions; an excerpt of this formalization is given in Appendix A.

A Lambdapi Implementation

We summarize here the Lambdapi development corresponding to our semantics. The first block defines universes, signatures, interpretation, and subtyping.

On top of this infrastructure, we define indexed vectors and their signatures (Figure **??**). The family `VectorSig` exposes both `empty` and `cons`, whereas `NonEmptyVector` exposes only `cons`, reflecting the constructor subset relation.

Finally, we can define concrete vectors and verify subtyping properties inside Lambdapi (Figure **??**). The assertions show that `NonEmptyVector` is a subtype of `VectorSig`, and that `head` requires no empty case, as predicted by our theory.

The complete Lambdapi sources, along with additional libraries and small proofs, are included in the accompanying repository.

B Conclusion

In this research, we introduced a compact yet powerful subtype system that demonstrates remarkable versatility and compatibility with various type theories. Our system effectively

addresses the challenge of reusing and subtyping constructors in indexed datatypes through the novel introduction of first-class datatype signatures. A particularly notable feature of our approach is its ability to eliminate trivial proof obligations even when an arbitrary set of constructors is involved, significantly reducing the proof burden in practical applications. We have illustrated the system's flexibility through several practical examples drawn from real-world programming scenarios. Furthermore, we have provided a concise implementation that maintains simplicity without sacrificing expressiveness, making our approach accessible for integration into existing proof assistants and type systems.

C Future Work

There are several directions in which we would like to extend this work.

- *Mechanized meta-theory.* We plan to formalize the typing and subtyping rules, together with subject reduction and progress, in a proof assistant such as Coq or Agda, and connect the proofs to our prototype implementation.
- *Larger case studies.* Our current examples are small; applying constructor subset signatures to larger developments in existing proof assistants (e.g. libraries of lists and vectors) would help validate their practical benefits.
- *Richer signatures.* Finally, we intend to explore simple extensions of signatures with basic set operations on constructor sets (such as unions) while keeping the meta-theory and implementation lightweight.

References

1. D. Aspinall and A. Compagnoni. Subtyping dependent types. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, pages 86–97, 1996. doi:10.1109/LICS.1996.561307.
2. Gilles Barthe. Order-sorted inductive types. *Information and Computation*, 149(1):42–76, 1999. URL: https://www.sciencedirect.com/science/article/pii/S0890540198927511, doi:10.1006/inco.1998.2751.
3. Gilles Barthe and Maria João Frade. Constructor subtyping. In *Proceedings of the 8th European Symposium on Programming Languages and Systems*, ESOP '99, page 109127, Berlin, Heidelberg, 1999. Springer-Verlag.
4. Matthew Fluet and Riccardo Pucella. Practical datatype specializations with phantom types and recursion schemes, 2005. arXiv:cs/0510074.
5. Maria João Frade. Type-based termination of recursive definitions and constructor subtyping in typed lambda calculi. 2003. URL: https://api.semanticscholar.org/CorpusID:115763806.
6. Christopher Jenkins, Colin McDonald, and Aaron Stump. Elaborating inductive definitions and course-of-values induction in cedille, 2019. arXiv:1903.08233.
7. Andrew Marmaduke, Christopher Jenkins, and Aaron Stump. Zero-cost constructor subtyping. In Olaf Chitil, editor, *IFL 2020: 32nd Symposium on Implementation and Application of Functional Languages, Virtual Event / Canterbury, UK, September 2-4, 2020*, pages 93–103. ACM, 2020. doi:10.1145/3462172.3462194.
8. Rodrigo Marques, Mário Florido, and Pedro Vasconcelos. Towards algebraic subtyping for extensible records, 2024. URL: https://arxiv.org/abs/2407.06747, arXiv:2407.06747.
9. Per Martin-Löf. Intuitionistic type theory by per martin-löf. notes by giovanni sambin of a series of lectures given in padova, june 1980. 2021.

10 Johan Nordlander. Polymorphic subtyping in o'haskell. *Science of Computer Programming*, 43(2-3):93–127, 2002. Validerad; 2002; 20070227 (ysko). doi:10.1016/S0167-6423(02)00026-6.

11 Ulf Norell. Towards a practical programming language based on dependent type theory. 2007. URL: https://api.semanticscholar.org/CorpusID:118357515.

12 Ronan Saillard. Typechecking in the lambda-pi-calculus modulo : Theory and practice. (vérification de typage pour le lambda-pi-calcul modulo : théorie et pratique). 2015. URL: https://api.semanticscholar.org/CorpusID:2853829.

```
// Core universes and kinds
constant symbol kind : TYPE;
constant symbol set1 : kind;
constant symbol nat : TYPE;
constant symbol Z : nat;
constant symbol S : nat  nat;

symbol typed : kind  TYPE;
rule typed set1  kind;

constant symbol unit : TYPE;
constant symbol u : unit;

// Indexed family of base kinds
constant symbol indexes_type : TYPE;
symbol static_symbol : indexes_type  kind;
symbol arrow : kind  kind  kind;

constant symbol sigma :  (P :  (k: kind), kind), TYPE;
constant symbol mk_sigma :
   (P :  (k: kind), kind) (k : kind),
    typed (P k)  sigma P;

// Indices for static_symbol
constant symbol index_null : indexes_type;
constant symbol index_succ : kind  indexes_type  indexes_type;

rule typed (static_symbol index_null)  kind;
rule typed (static_symbol (index_succ $k $res))
   (n : typed $k), typed (static_symbol $res);

// Signatures over constructors
symbol Constructors :  (k: kind), typed k  typed k  unit;
constant symbol signature : nat  kind;
constant symbol signature_bottom : typed (signature Z);
constant symbol signature_cons :
   (n : nat) (k : kind),
    typed k  typed (signature n)  typed (signature (S n));

symbol constructor_null :  (k : kind), kind;
symbol constructor_append : kind  kind  kind;

rule typed (constructor_null $k)  typed $k;
rule typed (constructor_append $k $a)
   (n : typed $k), typed $a;

// Interpret a signature as a kind (type)
symbol interpret :  (n : nat), typed (signature n)  kind;

// Injection into interpreted signatures
symbol inject :
   (n : nat) (sig : typed (signature (S n))) (base : kind),
    typed base  typed (interpret (S n) sig);

// Case type for a single constructor
symbol constructor_case_type : kind  kind  TYPE;
rule constructor_case_type (constructor_null $base) $Q  typed $Q;
rule constructor_case_type (constructor_append $arg $rest) $Q
   (x : typed $arg), constructor_case_type $rest $Q;

// Cases for all constructors in a signature
symbol match_cases :  (n : nat), typed (signature n)  kind  TYPE;
rule match_cases Z signature_bottom $Q  unit;
rule match_cases (S Z) (signature_cons Z $k $p signature_bottom) $Q
  constructor_case_type $k $Q;
rule match_cases (S (S $n)) (signature_cons (S $n) $k $p $rest) $Q
   (case : constructor_case_type $k $Q),
    match_cases (S $n) $rest $Q;

// Match on a value of interpreted signature type
symbol match_interpret :
   (n : nat) (sig : typed (signature n)) (Q : kind),
    typed (interpret n sig)
    match_cases n sig Q
    typed Q;

// Natural numbers as interpreted signature
symbol Nat : typed (static_symbol index_null);
symbol NatSig : typed (signature (S (S Z)));

// Internal Nat constructors
symbol z_base : typed Nat;
symbol succ_base : typed Nat  typed Nat;

// Public signature-typed constructors
symbol z : typed (interpret (S (S Z)) NatSig);
rule z  inject (S Z) NatSig Nat z_base;

symbol s : typed (interpret (S (S Z)) NatSig)  typed Nat;
symbol succ :
  typed (interpret (S (S Z)) NatSig)
  typed (interpret (S (S Z)) NatSig);
```

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

// VectorSig - signature family parameterized by element type and length
symbol VectorSig :
   (A : typed set1)
    (n : typed (interpret (S (S Z)) NatSig)),
      typed (signature (S (S Z)));

// Base constructors for vectors (internal use)
symbol empty_base :  (A : typed set1), typed (Vector A z);
symbol cons_base :
   (A : typed set1)
    (n : typed (interpret (S (S Z)) NatSig)),
      typed A
      typed (interpret (S (S Z)) (VectorSig A n))
      typed (Vector A (succ n));

// Define VectorSig with empty and cons constructors
rule VectorSig $A $n  signature_cons (S Z)
  (constructor_null (Vector $A z))
  (empty_base $A)
  (signature_cons Z
    (constructor_append
       $A
       (constructor_append
          (interpret (S (S Z)) (VectorSig $A $n))
          (constructor_null (Vector $A (succ $n)))))
    (cons_base $A $n)
    signature_bottom);

// Public constructors on signature-typed vectors
symbol empty :
   (A : typed set1),
    typed (interpret (S (S Z)) (VectorSig A z));
rule empty $A
  inject (S Z) (VectorSig $A z) (Vector $A z) (empty_base $A);

symbol cons :
   (A : typed set1)
    (n : typed (interpret (S (S Z)) NatSig)),
      typed A
      typed (interpret (S (S Z)) (VectorSig A n))
      typed (interpret (S (S Z)) (VectorSig A (succ n)));
rule cons $A $n $elem $vec
  inject (S Z)
         (VectorSig $A (succ $n))
         (Vector $A (succ $n))
         (cons_base $A $n $elem $vec);

// Helper to build a singleton-constructor signature
symbol mk_signature :
   (k : kind), typed k  typed (signature (S Z));
rule mk_signature $k $p
  signature_cons Z $k $p signature_bottom;

// NonEmptyVector - signature with only the cons constructor
symbol cons_vec_base :
   (A : typed set1)
    (n : typed (interpret (S (S Z)) NatSig)),
      typed A
      typed (interpret (S (S Z)) (VectorSig A n))
      typed (Vector A (succ n));

symbol NonEmptyVector :
   (A : typed set1)
    (n : typed (interpret (S (S Z)) NatSig)),
      typed (signature (S Z));
rule NonEmptyVector $A $n
  mk_signature
    (constructor_append
       $A
       (constructor_append
          (interpret (S (S Z)) (VectorSig $A $n))
          (constructor_null (Vector $A (succ $n)))))
    (cons_vec_base $A $n);

// Head function - only ONE case needed (no empty case!)
symbol head :
   (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
    ( elem vec, elem);
```

Listing 7 Indexed vectors and non-empty signatures

```
// 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 and subtyping assertions