Constructor subtyping with indexed types

Short paper

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. These signatures make the intended subset explicit at the type level, enabling pattern matching to rule out impossible branches without generating spurious obligations. Our prototype indicates that this approach can eliminate many trivial proof obligations while preserving type safety.

We implement our encoding in Lambdapi and test it against a fragment of the type system. Building on this implementation, we also develop a small proof assistant that uses constructor-subset subtyping.

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 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.

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 those values of T whose *outermost (head) constructor*, when inspected in canonical form, is drawn from Φ. This is the information that coverage checking for pattern matching consumes directly, and it is exactly what makes types such as "non-empty lists" inhabited (the tail may still be empty). 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.

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^*.

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. 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.

$$\textsc{Rule 1}\ \frac{\begin{array}{c}\Gamma \vdash T\ \Delta^* : \mathit{Type} \quad \Phi = (C_1, \ldots, C_n) \quad (C_1 \ldots C_n) \subseteq \mathcal{C}_{\text{all}} \\ \forall i,\ 1 \le i \le n,\ \Gamma \vdash C_i : \Delta_i \to T\ \Delta_i'^* \\ \forall i,\ 1 \le i \le n,\ \forall j,\ 1 \le j \le |\Delta^*|,\ \mathsf{AGAINST}(\Delta_j^*,\ \Delta_{i,j}'^*)\end{array}}{\Gamma \vdash \{T\Delta^* :: \Phi\} : \mathit{Type}}$$

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

$$\textsc{Rule 3}\ \frac{\begin{array}{c}\Gamma \vdash T\ \Delta^* : \mathit{Type} \quad \Phi = (C_1, \ldots, C_n) \\ \Gamma\ \vdash\ C : \Delta_C \to T\ \Delta_C'^* \quad C \in \Phi \qquad (T\ \Delta_C'^*)_{\beta\eta} = (T\ \Delta^*)_{\beta\eta}\end{array}}{\Gamma \vdash C\ \Delta_C : \{T\ \Delta^* :: \Phi\}}$$

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

$$\textsc{Rule 5}\ \frac{\Gamma \vdash T\ \Delta^* : \mathit{Type} \quad \Gamma \vdash F\ :\ \Delta \to A \qquad \Gamma \vdash A \sqsubseteq \{T\ \Delta^* :: \Phi'\}}{\Gamma \vdash F\ \Delta : \{T\ \Delta^* :: \Phi'\}}$$

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

Operational reading.

Under the usual canonical forms lemma for inductive values, a closed value of a signature type $\{T\ \Delta^* :: \Phi\}$ reduces (in weak head normal form) to a constructor application $C\ \overrightarrow{u}$ with $C \in \Phi$ and $(T\ \Delta_C'^*)_{\beta\eta} = (T\ \Delta^*)_{\beta\eta}$. Thus, signature types refine *which head constructors* can appear at a given program point, exactly the information required to remove unreachable branches in pattern matching.

Now we can introduce the $\mathsf{AGAINST}$ rules that try to generalize indexed datatypes between constructors. Operationally, $\mathsf{AGAINST}(p, t)$ can be read as a *first-order matching* or *generalization* check: the signature indices p (which may contain variables) must be general enough to accommodate the constructor result indices t. This is a predicate, so indexed values of different constructors have to respect these rules. The equality $=_\alpha$ simply means the alpha-equivalence relation.

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

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

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.

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

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

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

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

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

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 1** (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 $M = case\ v\ of\ Q\ \{C_i\ \Delta_i \Rightarrow N_i\ \Delta_i, \ldots\}$.* 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 $v : \{T :: \Phi\}$. There must be some pattern $C_i : \Delta_i$ that matches v: by Rule 3 we know that the head constructor of v lies in Φ, and by Rule 7 we have a corresponding branch for every constructor listed in Φ. 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 2** (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.

```
Inductive nat : TYPE :=
| _0 : nat
| +1 : nat → nat;

Inductive vec : Set → {nat | 0 | 1} → TYPE :=
| empty : Π (s : Set), vec s _0
| cons : Π (s : Set) (n : nat), vec s n → lift s → vec s (+1 n);

Def id_sig : Π (v : nat), {vec nat v | cons | empty} → {vec nat v | cons | empty} :=
  λ (v : nat) (m : {vec nat v | cons | empty}), m;

Def head_nonempty : Π (v : nat), {vec nat (+1 v) | cons} → {nat | 0 | +1} :=
  λ (v : nat) (m : {vec nat (+1 v) | cons}),
    match m as {vec nat (+1 v) | cons} return {nat | 0 | +1} with
    | cons k ⇒ (+1 _0)
    end;

Def vec_to_nonempty_append : Π (v : nat), {vec nat v | cons | empty} → nat → {vec nat (+1 v) | cons}
    :=
  λ (v : nat) (m : {vec nat v | cons | empty}) (x : nat), cons;
```

Figure 1 Minimal surface-language sample.

- *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 $M = \text{case } v \text{ of } Q\ \{\dots\}$ and $M' = \text{case } v' \text{ of } Q\ \{\dots\}$ 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 $M = \text{case } (C_i\ \Delta') \text{ of } Q\ \{\dots\}$ 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 Implementation

Our prototype type checker is implemented in Haskell on top of a dependently typed $\lambda\Pi$ calculus modulo rewriting [2]. 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. We use a transpiler to convert the fragment of lambdaPi to this fragment.

4 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.

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 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.

```
require open subtyping.encode;

symbol vec__sig_target : Set;
rule vec__sig_target ↪ vecLabel;
symbol vec__sig_indexes : indexes_type;
rule vec__sig_indexes ↪ index_poly_succ (index_dependent_lift_succ natLabel (index_null
    ));
symbol vec__sig_pattern_any : index_pattern_list;
rule vec__sig_pattern_any ↪ index_pattern_cons index_pat_var (index_pattern_cons
    index_pat_var (index_pattern_nil));
symbol vec__sig_phi_all : constructor_list;
rule vec__sig_phi_all ↪ constructor_list_cons emptyLabel (constructor_list_cons
    consLabel (constructor_list_nil));

symbol id_sig :
  Π (v : nat),
  (signature_term vec__sig_target
     (mk_delta_tel vec__sig_indexes vec__sig_pattern_any)
     (constructor_list_cons consLabel (constructor_list_cons emptyLabel (
    constructor_list_nil))))
  →
  (signature_term vec__sig_target
     (mk_delta_tel vec__sig_indexes vec__sig_pattern_any)
     (constructor_list_cons consLabel (constructor_list_cons emptyLabel (
    constructor_list_nil))));
rule id_sig ↪
  λ (v : nat) (m :
    (signature_term vec__sig_target
       (mk_delta_tel vec__sig_indexes vec__sig_pattern_any)
       (constructor_list_cons consLabel (constructor_list_cons emptyLabel (
    constructor_list_nil)))), m;

symbol head_nonempty :
  Π (v : nat),
  (signature_term vec__sig_target
     (mk_delta_tel vec__sig_indexes
       (index_pattern_cons index_pat_var
         (index_pattern_cons
           (index_pat_term
             (index_term_ctor (+1 _0) (index_term_cons index_term_var index_term_nil)))
           (index_pattern_nil))))
     (constructor_list_cons consLabel (constructor_list_nil)))
  → nat;
rule head_nonempty ↪
  λ (v : nat) (m :
    (signature_term vec__sig_target
       (mk_delta_tel vec__sig_indexes
         (index_pattern_cons index_pat_var
           (index_pattern_cons
             (index_pat_term
               (index_term_ctor (+1 _0) (index_term_cons index_term_var index_term_nil)
    ))
             (index_pattern_nil))))
       (constructor_list_cons consLabel (constructor_list_nil)))),
    (signature_case vec__sig_target
      (mk_delta_tel vec__sig_indexes
        (index_pattern_cons index_pat_var
          (index_pattern_cons
            (index_pat_term
              (index_term_ctor (+1 _0) (index_term_cons index_term_var index_term_nil))
    )
             (index_pattern_nil))))
      (constructor_list_cons consLabel (constructor_list_nil))
      natLabel
      m
      (case_branches_cons consLabel (constructor_list_nil) natLabel
        (λ (k : constructor_type consLabel), (+1 _0))
        (case_branches_nil natLabel)));

symbol vec_to_nonempty_append :
  Π (v : nat),
  (signature_term vec__sig_target
     (mk_delta_tel vec__sig_indexes vec__sig_pattern_any)
     (constructor_list_cons consLabel (constructor_list_cons emptyLabel (
    constructor_list_nil))))
  → nat →
  (signature_term vec__sig_target
     (mk_delta_tel vec__sig_indexes
       (index_pattern_cons index_pat_var
         (index_pattern_cons
           (index_pat_term
             (index_term_ctor (+1 _0) (index_term_cons index_term_var index_term_nil)))
           (index_pattern_nil))))
     (constructor_list_cons consLabel (constructor_list_nil)));
rule vec_to_nonempty_append ↪
  λ (v : nat)
    (m :
      (signature_term vec__sig_target
        (mk_delta_tel vec__sig_indexes vec__sig_pattern_any)
        (constructor_list_cons consLabel (constructor_list_cons emptyLabel (
    constructor_list_nil)))))
    (x : nat),
    (signature_intro vec__sig_target
```