

Constructor subtyping with indexed types

2 Anonymous author

3 Anonymous affiliation

4 Abstract

5 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.

11 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.

16 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.

19 **2012 ACM Subject Classification** Theory of computation → Program verification; Theory of computation → Type theory

21 **Keywords and phrases** Datatype, Type Constructors, Constructor Subtyping

22 **Digital Object Identifier** 10.4230/LIPIcs...

23 **1 Introduction**

24 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.

30 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.

37 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.

44 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



© **Anonymous author(s);**

licensed under Creative Commons License CC-BY 4.0



Leibniz International Proceedings in Informatics

LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

XX:2 Constructor subtyping with indexed types

47 empty constructor to obtain a standard list type. More recent work on zero-cost constructor
48 subtyping in Cedille [7, 6], on order-sorted inductive types and overloading [3, 2], and on
49 algebraic subtyping for extensible records [8] shows how rich subtyping disciplines can be
50 added to strongly typed systems. However, these systems are tailored to specific core calculi
51 or to non-dependent settings, and do not directly address the combination of constructor
52 subtyping with indexed families in a general dependently typed setting.

53 In this paper we focus on a simple but expressive fragment of constructor subtyping
54 aimed at eliminating trivial proof obligations. Rather than allowing arbitrary constructor
55 overloading, we work with *constructor subsets*: first-class types of the form $\{T :: \Phi\}$, where
56 T is an inductive family and Φ is a finite list of its constructors. Intuitively, $\{T :: \Phi\}$ denotes
57 those values of T whose *outermost (head) constructor*, when inspected in canonical form, is
58 drawn from Φ . This is the information that coverage checking for pattern matching consumes
59 directly, and it is exactly what makes types such as “non-empty lists” inhabited (the tail
60 may still be empty). We restrict attention to subtyping relations induced by inclusion of
61 constructor sets and by compatibility of the indices of T .

62 Our contributions are as follows.

- 63 ■ We define a subtyping discipline for constructor subsets as a conservative extension of a
64 $\lambda\Pi$ -style dependent type theory with inductive families. The system includes typing and
65 subtyping rules for constructor signatures and a dedicated elimination rule for pattern
66 matching on signature types.
- 67 ■ We show, through a series of examples, that constructor subsets capture common pro-
68 gramming idioms such as non-empty lists and simple invariants on inductive families,
69 while avoiding the proof obligations that arise when these invariants are expressed with
70 explicit propositions.
- 71 ■ We describe a two-layer prototype: a Lambdapi backend encoding of constructor signatures
72 and a Haskell transpiler that translates a lightweight surface fragment into signature-aware
73 LambdaPi terms. This architecture keeps programs concise while preserving a direct
74 correspondence with Rules 1–7.

75 Structure of the paper.

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

80 Many type systems are inspired by constructor subtyping to address constructor over-
81 loading, allowing the same constructor name to be shared by different datatypes, even in
82 pattern matching. However, its naive use can lead to problems with subject reduction, as
83 observed by Frade [5]. Subject reduction is the property that during the normalization of a
84 term its type is preserved; overloading constructors without care may violate this property.
85 For the sake of simplicity, in this work we do not consider overloading between distinct
86 datatypes. Instead we restrict subtyping to relations induced by subsets of constructors of a
87 single inductive family. We believe this fragment is already expressive enough to avoid the
88 trivial proof obligations discussed above, while keeping the meta-theory and implementation
89 comparatively simple.

```

1 Inductive list (A : Set) : Set :=
2   | cons : A → list A → list A
3   | empty : list A.
4
5 Definition head {A} (x : list A) :
6   x <> empty A → A :=
7   match x return x <> empty A → A with
8     | cons _ h l ⇒ fun _ ⇒ h
9     | empty _ ⇒
10      fun f ⇒
11        match (f eq_refl) with end
12 end.

```

Figure 1 Constructor exclusion in Coq with a proof obligation.

```

1 Inductive list' (A : Set) : bool → Set :=
2   | cons' : forall x, A →
3     list' A x → list' A true
4   | empty' : list' A false.
5
6 Definition head' {A} (x : list' A true) : A :=
7   match x with
8     | cons' _ _ h l ⇒ h
9   end.

```

Figure 2 Constructor exclusion in Coq using an indexed datatype.

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 $1 \leq i \leq n$. For a term T we use the side condition $T_{\beta\eta}$ to indicate that we inspect the $\beta\eta$ -normal form of T .

XX:4 Constructor subtyping with indexed types

We write \mathcal{C}_{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}_{all} ; in particular, mutually recursive definitions can be handled by first registering every constructor in \mathcal{C}_{all} and only then checking the rules below. A *signature* Φ is a finite list of constructor names drawn from \mathcal{C}_{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 $\text{AGAINST}(\Delta^*, \Delta'^*)$, defined later in this section, to ensure that indices in different constructors remain compatible. Because \mathcal{C}_{all} is finite, the side conditions that test membership in \mathcal{C}_{all} are decidable.

$$\begin{array}{c}
 \text{RULE 1 } \frac{\Gamma \vdash T \Delta^* : Type \quad \Phi = (C_1, \dots, C_n) \quad (C_1 \dots C_n) \subseteq \mathcal{C}_{\text{all}}}{\Gamma \vdash \{T \Delta^* :: \Phi\} : Type} \\
 \qquad \qquad \qquad \forall i, 1 \leq i \leq n, \Gamma \vdash C_i : \Delta_i \rightarrow T \Delta'^*_i \\
 \qquad \qquad \qquad \forall i, 1 \leq i \leq n, \forall j, 1 \leq j \leq |\Delta^*|, \text{AGAINST}(\Delta_j^*, \Delta'^*_{i,j}) \\
 \text{RULE 2 } \frac{\Gamma \vdash T \Delta^* : Type}{\Gamma \vdash \{T \Delta^* :: \Phi\} \sqsubseteq T \Delta^*} \\
 \\
 \text{RULE 3 } \frac{\Gamma \vdash T \Delta^* : Type \quad \Phi = (C_1, \dots, C_n) \quad \Gamma \vdash C : \Delta_C \rightarrow T \Delta'^*_C \quad C \in \Phi}{\Gamma \vdash \vec{u} : \Delta_C \quad (T \Delta'^*_C[\vec{u}/\Delta_C])_{\beta\eta} = (T \Delta^*)_{\beta\eta} \quad C \in \mathcal{C}_{\text{ALL}}} \\
 \qquad \qquad \qquad \Gamma \vdash C \vec{u} : \{T \Delta^* :: \Phi\} \\
 \\
 \text{RULE 4 } \frac{(T \Delta^*) =_{\beta\eta} (T' \Delta^*) \quad \Phi' \subseteq \Phi}{\Gamma \vdash \{T' \Delta^* :: \Phi'\} \sqsubseteq \{T \Delta^* :: \Phi\}} \\
 \\
 \text{RULE 6 } \frac{\Gamma \vdash A' \sqsubseteq A \quad \Gamma, x : A' \vdash B \sqsubseteq B'}{\Gamma \vdash (x : A) \rightarrow B \sqsubseteq (x : A') \rightarrow B'}
 \end{array}$$

116 Rule 5 (algorithmic only).

We keep the following rule as an implementation-level checking convenience (not part of the declarative calculus above). It is derivable from standard application typing plus subsumption:

$$\text{RULE 5 } \frac{\Gamma \vdash F : \Delta \rightarrow A \quad \Gamma \vdash \vec{u} : \Delta \quad \Gamma \vdash A[\vec{u}/\Delta] \sqsubseteq \{T \Delta^* :: \Phi\} \quad F \notin \mathcal{C}_{\text{ALL}}}{\Gamma \vdash F \vec{u} : \{T \Delta^* :: \Phi\}}$$

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

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

¹²⁶ **Operational reading.**

¹²⁷ Under the usual canonical forms lemma for inductive values, a closed value of a signature
¹²⁸ type $\{T \Delta^* :: \Phi\}$ reduces (in weak head normal form) to a constructor application $C \vec{u}$
¹²⁹ with $C \in \Phi$ and $(T \Delta_C^*[\vec{u}/\Delta_C])_{\beta\eta} = (T \Delta^*)_{\beta\eta}$. Thus, signature types refine *which head*
¹³⁰ *constructors* can appear at a given program point, exactly the information required to remove
¹³¹ unreachable branches in pattern matching.

¹³² The AGAINST rule ensures that indices can be properly generalized to avoid false positive
¹³³ subtyping that would result in empty types. Consider, for instance, a type signature T
¹³⁴ defined as:

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

¹³⁵ Such a type T leads to trivial proofs in dependent pattern matching unification—precisely
¹³⁶ what we aim to avoid, since T has no inhabitants, as can be trivially observed. More
¹³⁷ generally, any type constructor B can be represented as the bottom type \perp if it takes the
¹³⁸ form $\{B :: |\emptyset\}$.

¹³⁹ The intuitionistic logical explosion can be obtained through these types, commonly
¹⁴⁰ represented as $\Gamma, B \vdash A$, where A is any proposition [9]. It is important to note that while
¹⁴¹ $\{B :: |\emptyset\}$ is intuitionistically explosive, its subtype B remains inoffensive. This property
¹⁴² offers significant potential for representing proofs while preserving consistency, even when
¹⁴³ dealing with bottom types. We will discuss these properties in greater detail in the following
¹⁴⁴ sections.

¹⁴⁵ Now we can introduce the AGAINST rules that try to generalize indexed datatypes
¹⁴⁶ between constructors. Operationally, AGAINST(p, t) can be read as a *first-order matching* or
¹⁴⁷ *generalization* check: the signature indices p (which may contain variables) must be general
¹⁴⁸ enough to accommodate the constructor result indices t . This is a predicate, so indexed
¹⁴⁹ values of different constructors have to respect these rules. The equality $=_\alpha$ simply means
¹⁵⁰ the alpha-equivalence relation.

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

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

¹⁵¹ Rule 2 allows us to forget signature information at the same index instance: any value
¹⁵² of type $\{T \Delta^* :: \Phi'\}$ can be viewed as a value of the underlying type $T \Delta^*$. In this sense
¹⁵³ constructor subsets behave like *phantom types* [4]: the extra information carried by the
¹⁵⁴ signature is present at type-checking time but is not reflected in the runtime representation
¹⁵⁵ of values.

¹⁵⁶ Though phantom types are not widely known across programming languages—being
¹⁵⁷ primarily used in Haskell—they possess important properties for representing types that
¹⁵⁸ carry no information during runtime; that is, they are purely erased types [4]. In our system
¹⁵⁹ there is no dedicated elimination rule for pure phantom signatures, and Rule 7 only permits
¹⁶⁰ case analysis on signatures that still expose constructors.

XX:6 Constructor subtyping with indexed types

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

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

168 Remark (inductive well-formedness).

169 The previous paragraph sketches the kind of restrictions that an implementation must enforce
 170 so that inductive definitions remain well-founded (e.g. positivity/strictness conditions and
 171 well-scoped use of signatures in recursive arguments). Our prototype enforces a conservative
 172 check in the spirit of standard positivity conditions; however, we do not attempt to present a
 173 complete account of inductive well-formedness for signatures in this paper.

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

$$\text{RULE 7 } \frac{\Gamma \vdash T \Delta^* : \text{Type} \quad \Gamma \vdash Q : \text{Type} \quad \Phi = (C_1, \dots, C_n) \quad \Gamma \vdash M : \{T \Delta^* :: \Phi\} \\ \forall i, 1 \leq i \leq n, \Gamma \vdash C_i : \Delta_i \rightarrow T \Delta_i^{*} \quad \forall i, 1 \leq i \leq n, \Gamma \vdash N_i : \Delta_i \rightarrow Q}{\Gamma \vdash \text{case } M \text{ of } Q \{C_i \Delta_i \Rightarrow N_i \Delta_i, \dots\} : Q}$$

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

181 ▶ **Theorem 1** (Head-constructor refinement). *Let T be an inductive family with constructor
 182 set C , and let $S \subseteq C$. Define $D : T \rightarrow \text{Type}$ by inspecting only the head constructor of the
 183 canonical form: for each constructor application $C' \vec{u}$,*

$$184 D(C' \vec{u}) \equiv \begin{cases} \top & \text{if } C' \in S, \\ \perp & \text{if } C' \notin S. \end{cases}$$

185 Then there is a definable map

$$186 f : \{T :: S\} \rightarrow \Sigma(x : T), D(x).$$

187 Moreover, in any host theory that validates refinement introduction (from $x : T$ and $D(x)$ to
 188 $x : \{T :: S\}$), this map is part of an isomorphism $\{T :: S\} \simeq \Sigma(x : T), D(x)$.

189 **Proof sketch.** Define $f(x) := (x, \star)$. To justify $\star : D(x)$, use the canonical-forms property
 190 for signature types: if $x : \{T :: S\}$ is a value, its canonical form is $C \vec{u}$ with $C \in S$, hence
 191 $D(C \vec{u}) \equiv \top$.

192 For the converse direction, assume refinement introduction: from $(y, d) : \Sigma(x : T), D(x)$
 193 we obtain $y : \{T :: S\}$ and define $g(y, d) := y$. Then f and g are mutually inverse. ◀

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

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

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

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

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

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

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

- 199 ■ *Application* $M = M_1 \cdot M_2$. If either M_1 or M_2 is not a value, the induction hypothesis
 200 yields M'_i with $M_i \longrightarrow M'_i$, and we use the corresponding congruence rule $\xi\text{-app}_i$ to obtain
 201 a step for M . Otherwise both M_1 and M_2 are values. If M_1 is a lambda abstraction
 202 $\lambda x.N$, we can perform a β -reduction step. If M_1 is a constructor, then M is already a
 203 value (an applied constructor) and the conclusion holds.
- 204 ■ *Case expression* $M = \text{case } v \text{ of } Q \{C_i \Delta_i \Rightarrow N_i \Delta_i, \dots\}$. If v is not a value, the induction
 205 hypothesis gives v' with $v \longrightarrow v'$, and we use rule ξ -case. If v is a value, then by typing
 206 rule 7 we have $v : \{T \Delta^* :: \Phi\}$. There must be some pattern $C_i : \Delta_i$ that matches v :
 207 by Rule 3 we know that the head constructor of v lies in Φ , and by Rule 7 we have a
 208 corresponding branch for every constructor listed in Φ . In this situation we can apply
 209 rule ξ -case' and M takes a step.
- 210 ■ *Other forms.* For variables, constructors, and abstractions the canonical forms analysis
 211 from the typing rules shows that well-typed closed terms of signature type are either
 212 values or reduce by one of the cases above.

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

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

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

- 218 ■ *β -reduction.* If $M = (\lambda x.N) \cdot V$ and $M' = N[x := V]$, typing gives $\Gamma \vdash \lambda x.N : (x : A) \rightarrow B$
 219 and $\Gamma \vdash V : A$. By the substitution lemma we obtain $\Gamma \vdash N[x := V] : B[x := V]$, which
 220 is the same type as M .
- 221 ■ *$\xi\text{-app}_1$ and $\xi\text{-app}_2$.* In these cases we reduce a proper subterm of M . The induction
 222 hypothesis states that the reduced subterm preserves its type; reapplying the application
 223 typing rule reconstructs a typing derivation for M' with the same type R .
- 224 ■ *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
 225 induction hypothesis $\Gamma \vdash v' : \{T \Delta^* :: \Phi\}$, and Rule 7 then re-establishes $\Gamma \vdash M' : Q = R$.

XX:8 Constructor subtyping with indexed types

- 226 ■ ξ -case? In this case $M = \text{case } (C_i \Delta') \text{ of } Q \{\dots\}$ reduces to $M' = N_i \Delta'$. Typing rule 7
227 requires that for each branch we have $\Gamma \vdash N_i : \Delta_i \rightarrow Q$, and that $C_i : \Delta_i \rightarrow T \Delta^*$.
228 Instantiating with the actual arguments Δ' shows that M' has type Q , the same type as
229 M .
230 All other reduction rules follow the same pattern: either a proper subterm reduces and the
231 induction hypothesis applies, or a local computation such as β -reduction is justified by the
232 typing rules. In each case the result has the same type R as the original term. ◀

233 3 Case Examples

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

```
Term ::= Var(x) | App(f, a) | Lam(x, e) | Pi(x, A, B)
      | Constr(T,  $\vec{c}$ ) | Match(e, T,  $p \rightarrow e'$ ) | Notation(e, T)
```

237 □ **Figure 3** Term syntax of the example language

237 3.0.0.1 Mini-language overview.

238 Programs consist of a sequence of *static declarations* followed by regular statements. Static
239 declarations register constructors in \mathcal{C}_{all} (Rule 1), whereas dynamic statements bind names
240 to terms typed by the rules from Section 2. Terms use the usual λ -abstractions, function
241 types, applications, and pattern matches; the only novelty is that constructor signatures
242 $\{T :: \Phi\}$ appear explicitly in types. These informal descriptions suffice for the examples
243 below; the full concrete syntax is implemented in the prototype but omitted here.

244 We exploit the capability to operate with dynamically defined constructor subsets without
245 requiring trivial proof obligations. Since datatype signatures are first-class inhabitants in our
246 system, we can instantiate varying constructor sets through definition aliasing. The following
247 declarations register constructors for lists and expose two different signatures:

```
248 1 Static list : * > *.
249 2 Static empty : (A : *) -> (list A).
250 3 Static new : (A : *) -> A > {(list A) :: |new|empty} > (list A).
251 4 List |A :: * > * => {(list A) :: |new|empty}.
252 5 NonEmpty |A :: * > * => {(list A) :: |new|}.
```

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

255 Rule 3 ensures that each constructor inhabits the declared signature, while Rule 4 provides
256 $\text{NonEmpty } A \sqsubseteq \text{List } A$ because the latter merely adds the `empty` constructor. This formulation
257 enables the definition of functions that operate on both general lists and non-empty lists:

```
258 1 length |A ls :: (A : *) -> (List A) > Nat =>
259 2 [ls of Nat
260 3 |(empty _) => 0
261 4 |(new A head tail) => (+1 (length A tail))
262 5 ].
```

264

Listing 2 Length function compatible with both list variants

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

```

267 1 last |A ls :: (A : *) -> (NonEmpty A)> A =>
268 2 [ls of A
269 3   |(new A head tail) => [tail of A
270 4     |(empty _) => head
271 5     |(new A head2 tail2) => (last A (new A head2 tail2))
272 6   ]
273 7 ].
274 8 insertsort |xs v :: (List Nat)> Nat> (NonEmpty Nat) =>
275 9 [xs of (NonEmpty Nat)
276 10   |(empty _) => (new Nat v (empty Nat))
277 11   |(new _ head tail) => [(gte head v) of (NonEmpty Nat)
278 12     |false => (new Nat head (insertsort tail v))
279 13     |true => (new Nat v (new Nat head tail))
280 14   ]
281 15 ].
282 16 def list_inserted_has_a_last_element |xs v :: (List Nat)> Nat> Nat =>
283 17 (last Nat (insertsort xs v))
284 18
285 19
  
```

Listing 3 Subtyping application to eliminate trivial proof obligations

286 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`
 287 demonstrates that the sorted list inherits the non-empty signature with no auxiliary proofs—
 288 exactly the two obligations eliminated relative to Figure 1.

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

```

291 1 Static nat : *.
292 2 Static 0 : nat.
293 3 Static +1 : nat> nat.
294 4 Nat :: {nat :: |0 |+1}.
  
```

Listing 4 Attempt with phantom predecessor

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

```

301 1 Static nat : *.
302 2 Static 0 : nat.
303 3 Static +1 : Nat> nat.
304 4 Nat :: {nat :: |0 |+1}.
  
```

Listing 5 Recursive definition accepted by the algorithmic Rule 5 check

307 Because the predecessor now mentions the signature `Nat`, Rule 3 can derive the constructor
 308 typing judgment while the `AGAINST` predicate keeps indices aligned. This also demonstrates
 309 how $\{T :: \Phi\}$ can refer to itself without reintroducing the proof obligations we set out to
 310 eliminate [9].

XX:10 Constructor subtyping with indexed types

311 4 Implementation

312 Our implementation now has two components. The backend encoding in Lambdapi (`encode.lp`,
313 666 lines) defines signatures, constructor-list well-formedness, subtyping checks, and signature-
314 directed elimination on top of the $\lambda\Pi$ calculus modulo rewriting [12]. The frontend transpiler
315 (`Main.hs`, 956 lines) parses a restricted surface syntax and lowers it to backend terms.
316 Both components are in `subindex/subtyping/`. The lowering mirrors Rules 1–7 from Sec-
317 tion 2. Signature annotations become `signature_term`. Constructor introductions become
318 `signature_intro`. Signature-only matches become `signature_case`. Appendix A shows
319 representative input and generated output.

320 A Surface-to-Lambdapi Transpiler

321 The current artifact is organized around a translation pipeline: users write a compact surface
322 fragment, and `siglp-transpiler` lowers it to backend terms consumed by `subtyping.encode`.
323 This keeps examples short while preserving explicit generated code.

324 The supported surface fragment includes `Inductive` declarations, `Def` declarations, sig-
325 nature shorthand (`{T :: c_1 | ... | c_n}`), and signature-only `match` expressions. For
326 each inductive declaration, the transpiler generates helper symbols for target, index shape,
327 telescope, and constructor list. It rewrites term-level signatures to `signature_term`. It
328 rewrites constructor applications to `signature_intro`. It rewrites signature matches to
329 `signature_case`.

```
1 Static nat : *.
2 Static 0 : nat.
3 Static +1 : Nat -> nat.
4 Nat :: {nat :: |0 |+1}.
5
6 Static vec : * -> nat -> *.
7 Static empty : (s : *) -> (vec s 0).
8 Static cons : (s : *) -> (n : nat) -> (vec s n) -> (lift s) -> (vec s
    (+1 n)).
9
10 NonEmptyVecNat |v :: nat -> * => {(vec nat (+1 v)) :: |cons}.
11
12 head_nonempty |v m :: nat -> (NonEmptyVecNat v) -> Nat =>
13 [m of Nat
14   |(cons k) => (+1 0)
15 ].
```

■ **Figure 4** Mini-language rendering of the non-empty vector example

330 Compiling with `cabal run siglp-transpiler` (input: `examples/sample.siglp`, out-
331 put: `examples/sample.lp`) yields LambdaPi code such as Figure 5.

332 The complete generated files are omitted for space. Full examples are available in
333 `subindex/subtyping/transpiler/examples/`, and the target encoding used by generated
334 terms is `subindex/subtyping/encode.lp`.

```

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_delta : delta_tel;
rule vec_sig_delta ↪
  mk_delta_tel vec_sig_indexes vec_sig_pattern_any;

symbol head_nonempty : Π (v : nat),
  (signature_term vec_sig_target
    (mk_delta_tel vec_sig_indexes (...))
    (constructor_list_cons consLabel (constructor_list_nil))) → nat;

rule head_nonempty ↪ λ (v : nat) (m : ...),
  (signature_case vec_sig_target (mk_delta_tel vec_sig_indexes (...))
    (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)));

```

Figure 5 Generated LambdaPi excerpt after lowering signatures and matches

335 Backend shallow encoding (`encode.lp`).

336 The backend is organized as a shallow encoding of Rules 1–7 into LambdaPi symbols and re-
 337 write rules. Formation and well-formedness are captured by `signature_wfb` and `signature`.
 338 Subtyping uses boolean checkers (`signature_subtypeb`, `signature_to_base_subtypeb`)
 339 wrapped by `is_true` when a proposition is required. Elimination is represented di-
 340 rectly by `signature_term`, `case_branches`, and `signature_case`. Finally, a coercion hook
 341 (`coerce_rule`) inserts signature widening automatically when constructor-list inclusion is
 342 validated.

343 B Conclusion

344 In this research, we introduced a compact yet powerful subtype system that demonstrates
 345 remarkable versatility and compatibility with various type theories. Our system effectively
 346 addresses the challenge of reusing and subtyping constructors in indexed datatypes through
 347 the novel introduction of first-class datatype signatures. A particularly notable feature of
 348 our approach is its ability to eliminate trivial proof obligations even when an arbitrary set
 349 of constructors is involved, significantly reducing the proof burden in practical applications.
 350 We have illustrated the system’s flexibility through several practical examples drawn from
 351 real-world programming scenarios. Furthermore, we have provided a concise implementation
 352 that maintains simplicity without sacrificing expressiveness, making our approach accessible
 353 for integration into existing proof assistants and type systems.

354 C Future Work

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

- 356 ■ *Mechanized meta-theory.* We plan to formalize the typing and subtyping rules, together
 357 with subject reduction and progress, in a proof assistant such as Coq or Agda, and
 358 connect the proofs to our prototype implementation.

XX:12 Constructor subtyping with indexed types

```
symbol signature_wfb : Set → delta_tel → constructor_list → Bool;
inductive signature : Set → delta_tel → constructor_list → TYPE :=
| signature_nil : Π (T : Set) (g : delta_tel), signature T g constructor_list_nil
| signature_cons :
  Π (T : Set) (g : delta_tel) (c : labelConstructor) (cs : constructor_list),
  π (is_true (against_delta_b T g (constructor c))) →
  signature T g cs →
  signature T g (constructor_list_cons c cs);

symbol signature_subtypeb :
  Set → delta_tel → constructor_list →
  Set → delta_tel → constructor_list → Bool;

inductive signature_term : Set → delta_tel → constructor_list → TYPE :=
| signature_intro :
  Π (T : Set) (g : delta_tel) (phi : constructor_list) (c : labelConstructor),
  π (is_true (constructor_in_signature_b T g phi c)) →
  constructor_type c →
  signature_term T g phi;

symbol signature_case :
  Π (T : Set) (g : delta_tel) (phi : constructor_list) (Q : Set),
  signature_term T g phi →
  case_branches phi Q →
  lift Q;

coerce_rule coerce_signature
  (signature $T $g $phi1)
  (signature $T $g $phi2)
  $x ↦ coerce_signature $T $g $phi1 $phi2 $x;
```

Figure 6 Shallow backend encoding excerpt from encode.lp

- 359 ■ *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.
- 360 ■ *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.

365 References

- 366 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.
- 367 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.
- 368 3 Gilles Barthe and Maria João Frade. Constructor subtyping. In *Proceedings of the 8th European Symposium on Programming Languages and Systems*, ESOP ’99, page 109–127, Berlin, Heidelberg, 1999. Springer-Verlag.
- 369 4 Matthew Fluet and Riccardo Pucella. Practical datatype specializations with phantom types and recursion schemes, 2005. arXiv:cs/0510074.
- 370 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>.
- 371 6 Christopher Jenkins, Colin McDonald, and Aaron Stump. Elaborating inductive definitions and course-of-values induction in cedille, 2019. arXiv:1903.08233.
- 372 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.

- 385 8 Rodrigo Marques, Mário Florido, and Pedro Vasconcelos. Towards algebraic subtyping for
386 extensible records, 2024. URL: <https://arxiv.org/abs/2407.06747>, arXiv:2407.06747.
- 387 9 Per Martin-Löf. Intuitionistic type theory by per martin-löf. notes by giovanni sambin of a
388 series of lectures given in padova, june 1980. 2021.
- 389 10 Johan Nordlander. Polymorphic subtyping in o'haskell. *Science of Computer Programming*,
390 43(2-3):93–127, 2002. Validerad; 2002; 20070227 (ysko). doi:10.1016/S0167-6423(02)
391 00026-6.
- 392 11 Ulf Norell. Towards a practical programming language based on dependent type theory. 2007.
393 URL: <https://api.semanticscholar.org/CorpusID:118357515>.
- 394 12 Ronan Saillard. Typechecking in the lambda-pi-calculus modulo : Theory and practice.
395 (vérification de typage pour le lambda-pi-calcul modulo : théorie et pratique). 2015. URL:
396 <https://api.semanticscholar.org/CorpusID:2853829>.