Zero-Cost Constructor Subtyping

Andrew Marmaduke
The University of Iowa
Computer Science
Iowa City, IA, United States
andrew-marmaduke@uiowa.edu

Christopher Jenkins
The University of Iowa
Computer Science
Iowa City, IA, United States
christopher-jenkins@uiowa.edu

Aaron Stump
The University of Iowa
Computer Science
Iowa City, IA, United States
aaron-stump@uiowa.edu

ABSTRACT

Constructor subtyping is a form of subtyping where two inductive types can be related as long as the inductive signature of one is a subsignature of the other. To be a subsignature requires every constructor of the smaller datatype be present in the larger datatype (modulo subtyping of the constructors' types). In this paper, we describe a method for impredicatively encoding datatype signatures in Cedille (a dependently typed programming language) that supports highly flexible constructor subtyping, with the subtyping relation given by a derived notion of *type inclusion* witnessed by a heterogeneously typed identity function. Specifically, the conditions under which constructor subtyping is possible between datatypes are fully independent of the order in which constructors are listed in their declarations. After examining some extended case studies, we formulate *generically* a sufficient condition for constructor subtyping in Cedille using our technique.

CCS CONCEPTS

• **Theory of computation → Type theory**; *Lambda calculus*; • **Software and its engineering → Data types and structures**.

KEYWORDS

constructor subtyping, impredicative encodings, Cedille, generic programming

ACM Reference Format:
Andrew Marmaduke, Christopher Jenkins, and Aaron Stump. 2020. Zero-Cost Constructor Subtyping. In *IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages (IFL '20), September 2–4, 2020, Canterbury, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3462172.3462194

1 INTRODUCTION

Inductive datatypes are the least set of terms generated by their constructors. Constructor subtyping arises when we interpret the subtype relation as the subset relation between the sets of terms. Equivalently, one can interpret constructor subtyping as the subset relation between sets of constructors treated as uninterpreted constants. For example, the inductive datatype for natural numbers, $\mathbb{N}$, is represented by the set $\{0, succ\}$ and the inductive datatype for the (unquotiented) integers, $\mathbb{Z}$, is represented by the set $\{0, succ, pred\}$. It is trivial to see that $\{0, succ\} \subseteq \{0, succ, pred\}$ which implies $\mathbb{N} \subseteq \mathbb{Z}$.

Subtyping allows for function and proof reuse, and constructor subtyping in particular enriches the subtype relation to include relationships when an inductive datatype is a subset of any other inductive datatype. Function reuse is a natural use case of subtyping and although constructor subtyping is not required for reusing functions, many more kinds of instances of reuse are possible using a relation enriched by constructor subtyping. Additionally, constructor subtyping yields a form of incremental definition where a datatype is extended with additional constructors, implicitly inheriting the signature of the datatype being extended.

The concept of constructor subtyping is attributed to Kent Peterson and (independently) A. Salvesen by Coquand [8], but was developed by Barthe et al. [3, 4] into new calculi that directly support constructor subtyping. However, Barthe did not investigate constructor subtyping for indexed inductive datatypes. In this paper, we describe a highly flexible approach to constructor subtyping in the Cedille programming language, where the subtype relation is a derived notion of type inclusion — directly analogous to the set inclusion discussed earlier. In particular, this type inclusion yields a zero-cost coercion from a smaller type to a larger type in the subtyping relation.

By *zero-cost* we mean that there is no runtime penalty for coercing types in the subtyping relation. Indeed, the theory we work in is extrinsic and thus types can be viewed as the set of terms they classify. A type inclusion in this setting literally means that the set of classified terms for a smaller type is a subset of the set of classified terms for a larger type. Within Cedille, a type inclusion is realized by a heterogeneous typing for the identity function, e.g., an assignment of the type $\mathbb{N} \to \mathbb{Z}$ to $\lambda x.\, x$. For indexed types the presence of a (to be defined) implicit function type allows for type inclusions between lists and vectors enriched with additional constructors.

Precisely, our contributions are:

(1) a method of impredicative encoding of datatype signatures in Cedille that treats a list of constructors for a datatype as a label-indexed set, allowing users or language implementors their choice of labeling set (whose only requirement is decidable equality) and associated assignment of labels to constructors;
(2) a demonstration that this method supports highly flexible constructor subtyping, where the subtyping relation is a derived notion of type inclusion: for two compatible constructors to be identified, it is necessary only that they be assigned the same label;

IFL '20, September 2–4, 2020, Canterbury, United Kingdom

ACM ISBN 978-1-4503-8963-1/20/09…$15.00
https://doi.org/10.1145/3462172.3462194

$$\frac{\Gamma, x : T \vdash t' : T' \quad x \notin FV(|t'|)}{\Gamma \vdash \Lambda\, x{:}T.\, t' : \forall\, x{:}T.\, T'} \qquad \frac{\Gamma \vdash t : \forall\, x{:}T'.\, T \quad \Gamma \vdash t' : T'}{\Gamma \vdash t\ \text{-}t' : [t'/x]T}$$

$$|\Lambda\, x{:}T.\, t| \;=\; |t| \qquad |t\ \text{-}t'| \;=\; |t|$$

Figure 1: Implicit Functions

(3) and progress towards a surface-language implementation of datatypes with constructor subtyping, made by proving *generically* (for any labeling type and label-indexed family of constructor argument types) a sufficient condition for subtyping of datatypes.

All presented derivations, examples, and proofs are formalized in Cedille and available on github.

First, we provide the necessary background of Cedille's core theory and describe the features we use to implement constructor subtyping (Section 2). Next, we present the core idea behind the lambda encoding and describe the derivation in Cedille for natural numbers (Section 3). After, we explore three case studies involving parametric and indexed datatypes (Section 4). We then formulate generically a sufficient condition for subtyping of datatypes for the signatures produced by our proposed encoding (Section 5). The paper is concluded by remarking on related work (Section 6) and summarizing our results (Section 7).

2 BACKGROUND ON CEDILLE

Cedille is a dependently typed programming language whose core theory is called the Calculus of Dependent Lambda Eliminations (CDLE) [26]. It extends the extrinsically typed Calculus of Constructions (CoC) with three additional typing primitives: the implicit (or erased) function types of Miquel [23], the dependent intersection type of Kopylov [17], and an equality type of untyped terms. Critically, lambda-encoded datatypes supporting an induction principle are derivable in CDLE, which is not the case in CoC [14]. Moreover, efficient lambda encodings exist which alleviate prior concerns with lambda-encoded inductive data [11]. In the remainder of this section we review the three additional typing constructs that are added to CoC to form CDLE and mention two important derived constructs that are used in our encoding.

2.1 Implicit Functions and Erasure

Erasure in CDLE (denoted by vertical bars, e.g. $|t|$) defines what is operationally relevant in the theory. It can be understood as a kind of program extraction that produces untyped λ-terms. Typing information such as type abstractions or type annotations are all erased. Implicit functions, the rules for which are listed in Figure 1, give a way of expressing when a *term* should also be treated as operationally irrelevant.

We write a capital lambda to denote abstraction by either a type or an erased term (e.g. $\Lambda\, X.\, \Lambda\, y.\, \lambda\, x.\, x$ for type X and term y), a center dot for type application (e.g. $T_1 \cdot T_2$ or $t \cdot T$), a dash for erased-term application (e.g. $t_1\ \text{-}t_2$), and juxtaposition for term-to-term and type-to-term application (e.g. $t_1\ t_2$ and $T\ t$). In types, we use the standard forall quantifier symbol for both erased function types and type quantification (e.g. $\forall\, x{:}T_1.\, T_2$ and $\forall\, X{:}\star.\, T_2$). For convenience, we write an open arrow for an implicit function type that is not dependent (i.e. $T_1 \Rightarrow T_2$). In contrast, relevant dependent functions are written with the capital Greek pi (i.e. $\Pi\, x{:}T_1.\, T_2$) and a single arrow when not dependent (i.e. $T_1 \to T_2$). The typing rules for implicit functions are similar to those for ordinary ones, except for additional concerns of erasure. To introduce an implicit function, there is a syntactic restriction that the bound variable does not occur free in the erasure of the body of the function; this justifies the erasure of the elimination form, which completely removes the given argument.

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : [t_1/x]T_2 \quad |t_1| = |t_2|}{\Gamma \vdash [t_1, t_2] : \iota\, x{:}T_1.\, T_2}$$

$$\frac{\Gamma \vdash t : \iota\, x{:}T_1.\, T_2}{\Gamma \vdash t.1 : T_1} \qquad \frac{\Gamma \vdash t : \iota\, x{:}T_1.\, T_2}{\Gamma \vdash t.2 : [t.1/x]T_2}$$

$$|[t_1, t_2]| \;=\; |t_1| \quad |t.1| = |t| \quad |t.2| = |t|$$

Figure 2: Dependent Intersection

$$\frac{FV(t\ t') \subseteq dom(\Gamma)}{\Gamma \vdash \beta\{t'\} : \{t \simeq t\}} \qquad \frac{\Gamma \vdash t : \{t_1 \simeq t_2\} \quad \Gamma \vdash t' : [t_2/x]T}{\Gamma \vdash \rho\ t\ @\ x.T - t' : [t_1/x]T}$$

$$\frac{\Gamma \vdash t : \{t_1 \simeq t_2\} \quad \Gamma \vdash t_1 : T}{\Gamma \vdash \varphi\ t - t_1\ \{t_2\} : T} \qquad \frac{\Gamma \vdash t : \{\lambda\, x.\, \lambda\, y.\, x \simeq \lambda\, x.\, \lambda\, y.\, y\}}{\Gamma \vdash \delta - t : T}$$

$$|\beta\{t'\}| \;=\; |t'| \quad |\rho\ t\ @\ x.T - t'| \;=\; |t'|$$

$$|\varphi\ t - t_1\ \{t_2\}| \;=\; |t_2| \quad |\delta - t| \;=\; \lambda\, x.\, x$$

Figure 3: Equality

2.2 Dependent Intersections

In an extrinsically typed theory such as CDLE, terms do not have unique types. If we view all types as denoting sets of ($\beta\eta$-equivalence classes of) terms, then an intersection type is interpreted precisely as a set intersection. Additionally, this idea has a dependent counterpart appropriately called a dependent intersection, the rules for which are listed in Figure 2. Syntactically, the introduction form of a dependent intersection is a pair with the constraint that the components of the pair are $\beta\eta$-equal modulo erasure. This equality restriction on the components allows the erasure rule for the dependent intersection to forget one of the components, recovering our intuition for set intersection. We write the type of a dependent intersection with the Greek iota (i.e. $\iota\, x{:}T_1.\, T_2$), the introduction of dependent intersections with braces (i.e. $[t_1, t_2]$), and projections with a dot followed by a numeral for the first or second projection (i.e. $t.1$ or $t.2$). For convenience, we use the standard set intersection symbol (i.e. $T_1 \cap T_2$) for an intersection that is not dependent.

2.3 Equality

The propositional equality type of Cedille internalizes the judgemental $\beta\eta$-conversion (modulo erasure) of the theory. The rules

$$\frac{\Gamma \vdash f : S \to T \quad \Gamma \vdash t : \Pi\, x{:}S.\, \{f\ x \simeq x\}}{\Gamma \vdash \mathsf{intrCast}\ \text{-}f\ \text{-}t : \mathsf{Cast} \cdot S \cdot T}$$

$$\frac{\Gamma \vdash t : \mathsf{Cast} \cdot S \cdot T}{\Gamma \vdash \mathsf{cast}\ \text{-}t : S \to T}$$

$$|\mathsf{intrCast}\ \text{-}f\ \text{-}t| = \lambda\, x.\, x \qquad |\mathsf{cast}\ \text{-}t| = \lambda\, x.\, x$$

Figure 4: Type inclusions

governing it are given in Figure 3. Reflexive equalities are introduced with the β-axiom after a (potentially empty) series of rewrites (written with the Greek letter rho and a type guide to specify the position of the rewrite). The β-axiom allows for any well-scoped term to be used as the inhabitant of the equality. This, in combination with the fact that equality witnesses are erased from rewrites, makes the equality type effectively proof irrelevant. Additionally, the equality type has a strong form of the direct computation rule of Allen et al. [2], allowing a term's type to be changed to the type of another term if those two terms are provably equal. The direct computation rule is written with the Greek letter phi, typeset as φ. The δ-rule (written with the Greek letter delta) is used to distinguish unequal terms. It is a Principle of Explosion for the equality type from a single obviously false equality. In Cedille 1.1.2 some limited automation is included to schematize this rule for a larger set of trivially false equalities.

2.4 Type Inclusions

Capitalizing on CDLE's extrinsic typing, dependent intersections, and the direct computation law of the equality type, we now summarize how type inclusions are defined (see [16] for more details). For all types S and T, $\mathsf{Cast} \cdot S \cdot T$ is defined as the type of all functions f which are provably equal to the identity function:

$$\mathsf{Cast} \cdot S \cdot T = \iota\, f{:}S \to T.\, \{f \simeq \lambda\, x.\, x\}$$

For convenience, we present Cast axiomatically via a set of introduction, elimination, and erasure rules (Figure 4). However, note that these axiomatic rules are actually derivable in CDLE.

The introduction form $\mathsf{intrCast}\ \text{-}f\ \text{-}t$ takes as (operationally irrelevant) arguments a function f of type $S \to T$ and a proof t that, for all terms $x : S$, $f\ x$ is provably equal to x. The direct computation rule φ provides the justification for this rule: functions of type $S \to T$ that are merely extensionally equal to $\lambda\, x.\, x$ can be used to prove that $\lambda\, x.\, x$ *itself* has type $S \to T$. Operationally, using a witness t of the inclusion of a type S into T via the elimination form $\mathsf{cast}\ \text{-}t$ is then just an application of the identity function at type $S \to T$.

2.5 Strong Principle of Explosion

The Principle of Explosion trivializes a type theory in contexts where an absurdity is derivable. In CDLE, the false proposition, $\bot$, is defined in the standard way: $\bot = \forall\, X{:}\star.\, X$. However, a stronger version of the Principle of Explosion is possible in CDLE: instead of deriving some constant that is the inhabitant of all types, we are able to inhabit all types with *any* well-scoped term. This is a consequence of the axiom φ for equality:

$$\frac{\Gamma \vdash f : \bot \quad FV(t) \subseteq dom(\Gamma)}{\Gamma \vdash \mathsf{explode}\ \text{-}f\ t : T}$$

$$|\mathsf{explode}\ \text{-}f\ t| = |t|$$

Figure 5: Strong principle of explosion

$$\mathsf{explode} \cdot X\ \text{-}f\ t = \varphi\ (f \cdot \{f \simeq t\})\ -\ (f \cdot X)\ \{t\}$$

Figure 5 presents these concepts axiomatically. Note that, critically, the erasure of explode is the erasure of the well-scoped term t.

3 ORDER-INVARIANT LAMBDA ENCODINGS

We introduce a high-level syntax both for convenience in defining signature of inductive datatypes, and to illustrate how constructor subtyping might fit in the design of a surface language. The proposed syntax follows a similar style found in many functional languages. For example, the type of natural numbers is defined:

$$\mathtt{data}\ Nat : \star = zero : Nat \mid succ : Nat \to Nat$$

For constructor subtyping we focus on the following two operations on signatures: type extension and label equality constraints. Type extension allows a new datatype to be defined by extending a previously defined datatype with new constructors. For example, the type of integers is defined by extending the type of natural numbers:

$$\mathtt{data}\ Int\ \mathtt{extends}\ Nat\ \mathtt{with}\ pred : Int \to Int$$

With type extension, the type Int is defined with three constructors: $zero$, $succ$, and $pred$. Additionally, the shared constructors between Nat and Int are equal with respect to the underlying equality of the theory. Critically, this definition should make Nat a subtype of Int. Thus, any function argument that accepts an Int value should also accept a Nat value.

Equality constraints on labels allow for a more precise correspondence between constructors of datatypes. These constraints allow two types to agree on a shared subset of their constructors even though neither one is necessarily a subtype of the other. For example, a type of natural numbers starting instead at one but with a shared successor could be defined as:

$$\begin{aligned}&\mathtt{data}\ Nat1\ = succ : Nat1 \to Nat1 \mid one : Nat1\\ &\mathtt{where}\ Nat1.succ = Nat.succ\end{aligned}$$

Note that while type extension can be simulated by label equality constraints, the converse is not true. For instance, consider this artificial definition of a type with a single element:

$$\begin{aligned}&\mathtt{data}\ Nat2\ = succ : \bot \Rightarrow Nat2 \to Nat2 \mid unit : Nat2\\ &\mathtt{where}\ Nat2.succ = Nat.succ\end{aligned}$$

Although the type $Nat2$ semantically has only one inhabitant, the signature still correctly associates $Nat.succ$ and $Nat2.succ$ with the same label. Indeed, the erasures of these two constructors are identical because the $\bot$-argument to $Nat2.succ$ is erased. However, type extension (as described previously) can not be used to extend the *Unit* type (the canonical singleton type) to produce the types

Nat and $Nat2$ where the successor constructors are equal. That is, all of the constructors of two types can be equal without the types being equal (i.e. a mutual type inclusion). In the case of indexed datatypes this restriction becomes more apparent (and less artificial), as we will see in our case studies later in the paper. Thus, it is useful to have both constructs: type extensions and label equality constraints, because the first guarantees a type inclusion and the second gives enough expressivity for any desired relationship between constructors.

3.1 Deriving Inductive Datatypes

Since the Church encoding is the most familiar method of encoding inductive types, we choose it as our starting point for explaining our order-invariant encoding. We begin with a refresher on Church encodings and describe why constructor subtyping fails for inductive data that are derived from this encoding. After, we discuss how the situation can be amended to support constructor subtyping but still with Church-style folds.

The Church encoding of an inductive datatype identifies it with its iteration scheme. Thus, the interpretations for the constructors of the corresponding datatype are encoded as an ordered list of arguments to that scheme. For example, the type of Church encoded natural numbers is

$$CNat = \forall X : \star.\ \underbrace{X}_{\text{zero}} \to \underbrace{(X \to X)}_{\text{successor}} \to X$$

where the first input interprets zero and the second input interprets successor. The constructors are then defined by applying the corresponding interpretation of that constructor to the constructor's arguments, replacing recursive subdata with iteratively computed results. For instance, the successor function is defined in the following way:

$$succ\ n = \Lambda X.\, \lambda z.\, \lambda s.\, s\ (n \cdot X\ z\ s) \tag{1}$$

Suppose we wanted to define an integer type that was a supertype of the above defined natural number type. The naive approach would be to add a constructor for predecessor to the list:

$$CInt = \forall X : \star.\ \underbrace{X}_{\text{zero}} \to \underbrace{(X \to X)}_{\text{successor}} \to \underbrace{(X \to X)}_{\text{predecessor}} \to X$$

$$succ\ n = \Lambda X.\, \lambda z.\, \lambda s.\, \underline{\lambda p.}\, s\ (n \cdot X\ z\ s\ \underline{p}) \tag{2}$$

Unfortunately, this causes the definition of successor defined by (2) for the Church encoded integer type to be unequal to the natural number successor defined by (1) because of the additional lambda abstraction and application. Observe that constructors are being disambiguated by the order in which they appear in the Church encoded type definition. To implement constructor subtyping we need constructors to instead be disambiguated in an order-invariant way in the type signature.

Note that the function type $A \to B \to C$ is isomorphic to any reordering of the argument types (i.e. $A \to B \to C \simeq B \to A \to C$). This means that the constructors in Church encoded data can be reordered up to isomorphism, but this is not sufficient for zero-cost constructor subtyping. Instead, we need to be able to reorder up to type inclusion. To achieve this we utilize the power of intersection types. Indeed, the non-dependent variant of the intersection type is associative and commutative with respect to type inclusion. Thus, a first attempt at fixing the Church encoding is to replace the ordered list of constructors with an intersection of constructors:

$$CSNat = \forall X : \star.\ (\underbrace{X}_{\text{zero}} \cap \underbrace{(X \to X)}_{\text{successor}}) \to X$$

However, now a fold requires that the interpretation of zero and the interpretation of successor are equal, which would severely compromise the utility of this type as an encoding of natural numbers. By removing the order we have eliminated a method for disambiguating the constructors which is compensated for in the type theory by demanding that the constructors are equal.

With that in mind, we pick a type L with decidable equality to represent constructor labels, where we intend that different label values will be used to disambiguate different constructors. Now, the label type is used as an indexing set to the set of constructors with each constructor disambiguated via an additional erased argument that checks if the indexed value is equal to the associated label.

$$\begin{aligned} CSNatPack \cdot X\ \ell = &(\{\mathsf{lzero} \simeq \ell\} \Rightarrow X) \\ &\cap (\{\mathsf{lsucc} \simeq \ell\} \Rightarrow X \to X) \\ CSNat = \forall X : \star.\ &(\Pi\, \ell : \mathsf{L}.\, CSNatPack \cdot X\ \ell) \to X \end{aligned}$$

There are two important reasons why this encoding works. First, in the event that an equality is false it will be equivalent to $\bot$. Thus, the strong Principle of Explosion allows components of the intersection whose label equality is false to be equal to any well-scope term and in particular to a constructor term whose label equality is true. Without the strong Principle of Explosion the encoding would not be inhabitable for any datatype with more than one constructor.

Second, the constructors can be made disjoint by picking a unique label for every constructor. In fact they *must* be disjoint, as otherwise a fold could not decide between two given possible constructors. When computing a fold over this encoding the supplied function can match on the label (because it has decidable equality) to obtain the correct constructor, but only if the mapping from labels to constructors is unique. Now an extension to $CSNat$ is as simple as adding a constructor into the intersection with an appropriate disjoint selection of a label. Indeed, this concept is applicable to other encodings and more generally for any label-indexed signature functor, as we show in Section 5.

Two outstanding questions about this definition remain: what is the overhead, and how should labels be assigned to constructors. First, the packaging only imposes one layer of indirection in the definition. However, it does require inspecting the label value to produce the correct constructor when performing a fold. This inspection can always be defined as a sequence of equality comparisons on the occurring labels in the signature. With this implementation the cost is the same as the number of constructors times the cost of comparing labels for equality. In our formalization, as a matter of convenience, we use a natural number type to represent labels which has a linear time cost to compare for equality. This means that, for an implementation, the choice of a label type can not be made lightly. With lambda-encoded data, the cost of equality can be made logarithmic by using a tree structure for the label types, but in a more mature language (such as Idris, Agda, or Coq) the standard natural number type may be internally represented more

efficiently. Additionally, a type represented by machine words may also be present, which would be the ideal label type (assuming that a limit of 2^{64} constructors is acceptable). If we assume that the cost of equality between labels is a `memcmp` between words then the cost of unpacking is proportional to the number of constructors for that datatype.

The second outstanding issue is that the method of assignment of labels has a few variations, each of which have different trade-offs. The obvious assignment is to give every constructor a unique label *except* when the user specifies when two constructors should be equal. Another variation is to assign labels based on the order the constructors are listed in the declaration. This is, coincidentally, how subtyping between datatypes works in Cedille as of version 1.1.2, and why zero-cost reuse between certain inductive datatypes with the same number of constructors can be manually derived [9]. There is no best choice about how labels should be assigned to constructors, but the method should be coherent and predictable.

The examples from this and the next section are formalized manually but one would hope that the encoding could be heavily abstracted such that only the inductive signature is necessary. Moreover, we use a Church-style encoding which is known to be inefficient when computing predecessors. To remedy this, in Section 5 we detail a generic development of a Mendler-style encoding.

4 CASE STUDIES

4.1 Naturals and Integers

We return to our example of Nat and Int as a warm-up for the exposition of our proposed encoding. Recall the definition of $CSNat$:

$$\begin{aligned} CSNatPack \cdot X\ \ell &= (\{\mathsf{lzero} \simeq \ell\} \Rightarrow X) \\ &\quad \cap (\{\mathsf{lsucc} \simeq \ell\} \Rightarrow X \to X) \\ CSNat &= \forall X{:}\star.\ (\Pi\, \ell{:}\mathsf{L}.\ CSNatPack \cdot X\ \ell) \to X \end{aligned}$$

Using this definition we are able to define the successor constructor by picking the correct label, projecting out of the intersection, and applying the arguments.

$$\begin{aligned} succ\ n = &\Lambda X.\, \lambda f.\, \mathsf{let}\ s = (f\ \mathsf{lsucc}).\underline{2}\ \text{-}\beta \\ &\mathsf{in}\ s\ (n\ f) \end{aligned} \tag{3}$$

Note that the underlined section, the projection out of the intersection type, is computationally irrelevant.

A $CSInt$ type can be defined merely by adding the predecessor constructor to the intersection.

$$\begin{aligned} CSIntPack \cdot X\ \ell &= (\{\mathsf{lzero} \simeq \ell\} \Rightarrow X) \\ &\quad \cap (\{\mathsf{lsucc} \simeq \ell\} \Rightarrow X \to X) \\ &\quad \cap (\{\mathsf{lpred} \simeq \ell\} \Rightarrow X \to X) \\ CSInt &= \forall X{:}\star.\ (\Pi\, \ell{:}\mathsf{L}.\ CSIntPack \cdot X\ \ell) \to X \end{aligned}$$

Moreover, the definition of successor is exactly the same *except* for projection out of the intersection.

$$\begin{aligned} succ\ n = &\Lambda X.\, \lambda f.\, \mathsf{let}\ s = (f\ \mathsf{lsucc}).\underline{2.1}\ \text{-}\beta \\ &\mathsf{in}\ s\ (n\ f) \end{aligned} \tag{4}$$

Like the definition in (3) the underlined projection in (4) is erased, but the two definitions are identical otherwise! Therefore, the erasures of these definitions are α-convertible. Indeed, without much effort we can prove that there is a type inclusion from $CSNat$ to $CSInt$ as desired.

Theorem 4.1. *The identity function has type* $CSNat \to CSInt$.

Proof. Note that here we supply a proof sketch and remind the reader that all examples and proofs have been formalized in Cedille and is available on github. First, note that it is enough to construct a function $f : CSNat \to CSInt$ that is extensionally equal to the identity function. The direct computation rule, φ, is then used to demonstrate that the identity function $(\lambda\, x.\, x)$ also has this type. Let

$$f\ n = CSNat.fold\ n\ CSInt.zero\ CSInt.succ$$

To prove that f is extensionally the identity we induct [1] on n . The base case is trivial because $CSNat.zero$ is identical to $CSInt.zero$. In the step case suppose that $n = CSNat.succ\ x$ and that $f\ x = (\lambda\, z.\, z)\ x$. The successor function commutes with the fold, thus $f\ (CSNat.succ\ x) = CSInt.succ\ (f\ x)$. Now the inductive hypothesis and our prior observation that $CSNat.succ = CSInt.succ$ concludes the proof. □

As an aside, the predecessor function is of course unequal to any other constructor of $CSNat$ because the associated label, which is computationally relevant in the constructor, is different from any other label used in the definition of $CSNat$.

4.2 Lists and Vector Trees

Zero-cost reuse between lists and vectors (i.e., length-indexed lists) is already possible in the current version (1.1.2) of Cedille — in both directions [9]. The direction of reuse from lists to vectors requires a dependent form of casts which demonstrates additional difficulties that may arise with defining constructor subtyping with a syntactic calculus as done by Barthe [4].

We define lists and vector trees using the higher level syntax introduced at the beginning of Section 3.

$$\mathtt{data}\ List\ (A : \star) : \star = nil : List \mid cons : A \to List \to List$$

Note that the parameter, A, is not applied to the type $List$ in the constructors intentionally. To handle type parameters the packaging type of the order-invariant lambda encoding must take an additional type argument. There is no other change to the underlying encoding aside from adding the type parameter.

A vector type is a length indexed list, and because the length value can be erased (since, if needed at runtime, it can be dynamically computed) it is clear from both prior work on re-use and from earlier developments in this paper that the constructors for the types will be equal. To add a constructor subtyping twist to the example, we consider a *vector tree* type which is a regular vector type extended with a branching constructor:

$$\begin{aligned} &\mathtt{data}\ VecTree\ (A : \star) : \mathbb{N} \to \star = \\ &\mid nil : VecTree\ 0 \\ &\mid cons : \forall\, n{:}\mathbb{N}.\ A \to VecTree\ n \to VecTree\ (n+1) \\ &\mid branch : \forall\, a, b{:}\mathbb{N}.\ VecTree\ a \to VecTree\ b \to VecTree\ (a+b) \\ &\mathtt{where}\ List.nil = VecTree.nil \mid List.cons = VecTree.cons \end{aligned}$$

[1] Induction is derived for the Church encoded data in the formalization following the standard approach from past work by Stump [27].

Note that in our prior discussion of this high level syntax we mentioned that type extension is a strictly weaker form of type definition for constructor subtyping. With this example it becomes clear why: using type extension to define $VecTree$, the elaboration of the inductive datatype would be required to make an arbitrary choice about the index for the *nil* and the *cons* constructors. In contrast, checking the equalities of $List.cons$ and $VecTree.cons$ amounts to checking $\beta\eta$-equality for their elaborations modulo erasure, which does not require arbitrary choices.

To prove that there is a cast from $List$ to $VecTree$ we need a dependent variant of type inclusion.

$$\mathsf{DepCast} \cdot A \cdot B = \iota\, f : \Pi\, a : A.\, B\ a.\, \{f \simeq \lambda\, x.\, x\}$$

Now the desired cast property can be expressed in the following way:

$$\mathsf{DepCast} \cdot (List \cdot A) \cdot (\lambda\, l.\, VecTree \cdot A\ (length\ l))$$

where $length$ is a function computing the length of a $List$.

Theorem 4.2. *The identity function has type*

$$\Pi\, l : List \cdot A.\, VecTree \cdot A\ (length\ \ell)$$

for any A.

Proof. As before it is enough to construct a function

$$f : \Pi\, l : List \cdot A.\, VecTree \cdot A\ (length\ l)$$

that is extensionally equal to the identity function. Unlike with $CSNat$ and $CSInt$, the dependency on the length of l requires that we use induction (as opposed to merely iteration) in the definition of f *itself* so that we can compute the correct length of the $VecTree$ in the inductive step. Thus, we define f as follows:

$$f\ l = List.induct\ l\ (V.nil \cdot A)\ (\Lambda\, m.\, V.cons\ \text{-}(length\ m))$$

where $V = VecTree$. Note that the use of induction is operationally equivalent to a fold. Thus, just as in the prior proof of $CSNat$ and $CSInt$, a straightforward induction on l is sufficient to prove that f is extensionally equal to the identity. □

4.3 Language Extensions

In this subsection we study yet another example of indexed inductive datatypes: one for the simply typed λ-calculus, and one for an extension of that calculus with numerals and addition. To derive an inductive type for the simply typed λ-calculus, we first define an auxiliary type encoding the internal simple types with two constructors:

$$\mathtt{data}\ Typ : \star = base : Typ \mid arr : Typ \to Typ \to Typ$$

Now we are able to define the simply typed λ-calculus:

$$\begin{aligned}
&\mathtt{data}\ Stlc : List \cdot Typ \to Typ \to \star = \\
&\mid var : \forall\, \Gamma : List \cdot Typ.\, \forall\, T : Typ. \\
&\qquad \Pi\, i : \mathbb{N}.\, \{at\ i\ \Gamma\ T\ \simeq\ \mathtt{tt}\} \Rightarrow Stlc\ \Gamma\ T \\
&\mid abs : \forall\, \Gamma : List \cdot Typ.\, \forall\, A : Typ.\, \forall\, B : Typ. \\
&\qquad Stlc\ (cons\ A\ \Gamma)\ B \to Stlc\ \Gamma\ (arr\ A\ B) \\
&\mid app : \forall\, \Gamma : List \cdot Typ.\, \forall\, A : Typ.\, \forall\, B : Typ. \\
&\qquad Stlc\ \Gamma\ (arr\ A\ B) \to Stlc\ \Gamma\ A \to Stlc\ \Gamma\ B
\end{aligned}$$

where *at* tests if a given element is at the ith depth of a list.

In order to extend this language with numerals, Typ must first be extended with an encoded type of numerals:

$$\mathtt{data}\ ETyp\ \mathtt{extends}\ Typ\ \mathtt{with}\ nat : ETyp$$

where $ETyp$ stands for "extended-Typ." Finally, we extend $Stlc$ with two constructors for numerals and a primitive addition function.

$$\begin{aligned}
&\mathtt{data}\ EStlc : List \cdot ETyp \to ETyp \to \star = \\
&\mid var : \forall\, \Gamma : List \cdot ETyp.\, \forall\, T : ETyp. \\
&\qquad \Pi\, i : \mathbb{N}.\, \{at\ i\ \Gamma\ T\ \simeq\ \mathtt{tt}\} \Rightarrow EStlc\ \Gamma\ T \\
&\mid abs : \forall\, \Gamma : List \cdot ETyp.\, \forall\, A : ETyp.\, \forall\, B : ETyp. \\
&\qquad EStlc\ (cons\ A\ \Gamma)\ B \to EStlc\ \Gamma\ (arr\ A\ B) \\
&\mid app : \forall\, \Gamma : List \cdot ETyp.\, \forall\, A : ETyp.\, \forall\, B : ETyp. \\
&\qquad EStlc\ \Gamma\ (arr\ A\ B) \to EStlc\ \Gamma\ A \to EStlc\ \Gamma\ B \\
&\mid num : \Pi\, \Gamma : List \cdot ETyp.\, \mathbb{N} \to EStlc\ \Gamma\ nat \\
&\mid add : \Pi\, \Gamma : List \cdot ETyp. \\
&\qquad EStlc\ \Gamma\ nat \to EStlc\ \Gamma\ nat \to EStlc\ \Gamma\ nat \\
&\mathtt{where}\ Stlc.var = EStlc.var \\
&\mid Stlc.abs = EStlc.abs \\
&\mid Stlc.app = EStlc.app
\end{aligned}$$

Notice that every occurrence of Typ in the definition of $Stlc$ is replaced instead with the more general type $ETyp$. This allows for numeral abstractions and higher-order numeral functions as expected in an extension to the simply typed λ-calculus. It is also another example of how equality constraints on constructor labels provides greater flexibility than type extension.

There is no novel technique in constructing a type inclusion between $Stlc$ and $EStlc$ except in writing down the correct function type between them. Suppose that $g : Typ \to ETyp$ and $h : List \cdot Typ \to List \cdot ETyp$ are equal to the identity function. Note that constructing g is similar to what we previously showed for $CSNat$ and $CSInt$. Once g is constructed, h is constructed by induction and then direct computation by φ. With g and h the correct type is

$$Stlc\ \Gamma\ A \to EStlc\ (h\ \Gamma)\ (g\ A)$$

for any context Γ and type A.

5 GENERIC SUBTYPING FOR INDUCTIVE DATATYPES

In the preceding case studies, we have used a surface language for constructor subtyping in which users mark explicitly the constructor labels they wish to identify. For example,

$$\begin{aligned}
&\mathtt{data}\ Int : \star = \\
&\mid izero : Int \\
&\mid isucc : Int \to Int \\
&\mid ipred : Int \to Int \\
&\mathtt{where}\ Nat.zero = Int.izero \mid Nat.succ = Int.isucc
\end{aligned}$$

indicates that the labels for constructors *zero* and *izero*, and *succ* and *isucc*, should be the same. This surface language has been informally justified by the method of encoding described in Section 3.1. However, an implementation must take the set of equations between labels given in the declaration and use some criterion to confirm

$$\begin{array}{rcl} \mathsf{Mono} \cdot F & = & \forall X{:}\star.\, \forall Y{:}\star. \\ & & \mathsf{Cast} \cdot X \cdot Y \to \mathsf{Cast} \cdot (F \cdot X) \cdot (F \cdot Y) \\ \mathsf{Alg} \cdot F \cdot T & = & \forall R{:}\star.\, (R \to T) \to F \cdot R \to T \end{array}$$

$$\frac{F : \star \to \star}{\mu F : \star} \qquad \frac{t_1 : \mathsf{Alg} \cdot F \cdot T \quad t_2 : \mu F}{\mathsf{fold}\ t_1\ t_2 : T}$$

$$\frac{F : \star \to \star \quad m : \mathsf{Mono} \cdot F \quad t : F \cdot \mu F}{\mathsf{in}\text{-}m\ t : \mu F}$$

$$\frac{F : \star \to \star \quad m : \mathsf{Mono} \cdot F \quad t : \mu F}{\mathsf{out}\text{-}m\ t : F \cdot \mu F}$$

$$\begin{array}{rcl} |\mathsf{out}\text{-}m\ (\mathsf{in}\text{-}m\ t)| & = & |t| \\ |\mathsf{fold}\ t_1\ (\mathsf{in}\text{-}m\ t_2)| & = & |t_1\ (\mathsf{fold}\ t_1)\ t_2| \end{array}$$

Figure 6: Interface for the framework of Firsov et al. [11]

that the corresponding constructors *themselves* may be equated before constructor subtyping can be supported. To ensure soundness of this criterion, we desire that the implementation produce evidence of subtyping between underlying datatype encodings.

In this section, we take a step towards an implementation. Specifically, we demonstrate generically a sufficient condition for establishing subtyping between encodings of datatypes with labeled constructors. This condition is formulated over datatype signatures, meaning that no inductive proof over the datatypes themselves is required to determine whether zero-cost constructor subtyping between them is possible.

The results of this section use two generic frameworks written in Cedille: the efficient impredicative encodings for datatypes with induction by Firsov et al. [11], and the work of Diehl et al. [9] for zero-cost program and data reuse for these encodings. Following a review of these works, we give an over-approximation Sig of signatures of datatypes with labeled constructors and prove a sufficient condition for subtyping between datatypes whose signatures are given by Sig. Finally, we illustrate how these generic results could correspond to surface-language constructs for data and program reuse, returning to the example of constructor subtyping for naturals and integers to show how to extend the definition of addition over naturals to integers in a style similar to the *datatypes á la carte* approach of Swierstra [28].

5.1 Review: Generic Mendler-style Encoding

Firsov et al. [11] provide a generic framework for efficient inductive impredicative encodings of datatypes in Cedille. By *generic*, what is meant is *parametric*: their development is parameterized by a covariant type scheme $F : \star \to \star$ giving the datatype signature. We review the definitions of this framework that we use, listed in Figure 6.

- Mono, the property that a type scheme is covariant (more precisely, *monotonic* with respect to the preorder on types induced by Cast);
- Alg, the generic shape of recursive definitions of functions over datatype μF that fall into the pattern of Mendler-style iteration;
- μ, which forms a datatype given a signature $F : \star \to \star$;
- fold, the recursive combinator for Mendler-style iteration.
- in, the generic datatype constructor; and
- out, the generic datatype destructor.

For the reader familiar with the framework of Firsov et al., we note that some transliteration is required: their Id is equivalent to our Cast, and their IdMapping is equivalent to our Mono.

The generic constructor and monotonicity. The datatype μF can be understood as the least fixedpoint of the type scheme F (a result known as *Lambek's lemma* [18], which Firsov et al. [11] proved holds for their encoding). It is well-known that unrestricted fixedpoint types in type theory lead to non-termination and inconsistency when the theory is interpreted as a logic under the *Curry-Howard correspondence*. To avoid such issues, the formation, introduction, or elimination of fixedpoint types must be restricted.

Usually, the restriction is the syntactic condition on the formation of the type μF that F is a positive type scheme, i.e., F is of the form $\lambda X : \star.\, T$ and in T the type variable X never occurs to the left of an odd number of arrow type constructors. Here, the restriction is on the constructor in and destructor out, and it is to require the existence of a *monotonicity witness* for F in the style of Matthes [20]. In the case of in, when t is an F-collection of μF predecessors, and m is a proof that F is monotonic, then in *-m t* constructs the successor value of type μF. The type of m, $\mathsf{Mono} \cdot F$, is the definition of monotonicity in the preorder whose underlying set is the set of CDLE types and whose ordering relation is type inclusions, Cast.

Mendler-style iteration. Mendler-style inductive types, first proposed by Mendler [21, 22] for impredicative encodings of datatypes in polymorphic type theory, provide an alternative to the conventional initial F-algebra semantics for inductive types (c.f. [29] for the categorical account). Roughly, the key difference between the conventional and Mendler-style formulation is that the latter introduces higher-rank polymorphism and higher-order functions. Starting simply, compare the Mendler-style encoding of naturals below to the familiar Church encoding:

$$\mathit{MNat} = \forall X{:}\star.\, X \to \underbrace{(\forall R{:}\star.\, (R \to X) \to R \to X)}_{\text{successor}} \to X$$

It is helpful to read the universally quantified type variable R as standing in for recursive occurrences of the type *MNat* itself. Thus, the interpretation of successor is as a polymorphic higher-order function taking a handle for making recursive calls $(R \to X)$ on predecessors, a predecessor of type R, and returns the appropriate next result. The polymorphic typing ensures that the interpretation of successor *cannot make recursive calls on arbitrary terms of type MNat*, and helps to explain why Mendler-style recursion schemes are guaranteed to be terminating when general recursion for natural numbers (which has a similar shape) is not.

The type family Alg gives the generic shape of the types of functions used in Mendler-style iteration. A function of type $\mathsf{Alg} \cdot F \cdot T$ is given an F-collection of datatype predecessors at the universally

$$\frac{\Gamma \vdash T_1 : \star \quad \Gamma, x{:}T_1 : T_2 : \star}{\Gamma \vdash \Sigma\, x{:}T_1.\, T_2 : \star}$$

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : [t_1/x]T_2}{\Gamma \vdash (t_1, t_2) : \Sigma\, x{:}T_1.\, T_2}$$

$$\frac{\Gamma \vdash t : \Sigma\, x{:}T_1.\, T_2 \quad \Gamma \vdash f : \Pi\, x{:}T_1.\, \Pi\, y{:}T_2.\, P\ (t_1, t_2)}{\Gamma \vdash \Sigma\text{-elim}\ t\ f : P\ t}$$

Figure 7: Dependent pair types

quantified type R, and a handle of type $R \to T$ for making recursive calls. Again, the polymorphic typing ensures only well-founded recursive calls are well-typed. For the typing rule of the generic recursive combinator fold for Mendler-style iteration, we take some $t_1 : \mathsf{Alg} \cdot F \cdot T$ and data $t_2 : \mu F$; for the computation rule at the bottom of Figure 6, fold t_1 acts on a value constructed with in by calling t_1 with a handle fold t_1 for recursive calls on the subdata.

5.2 Signature Containment

We now summarize in Proposition 5.2 the result of Diehl et al. [9] we use in our generic development. First, we give a precise definition of the signature containment relation, SigSub, for first-order datatype signatures. Diehl et al. call this relation an "identity algebra," and the idea behind it can be traced back to Hinze [15] and Abel et al. [1] where it is generalized to higher-order type schemes (but formulated in terms of function types, not type inclusions).

Definition 5.1 (Signature containment). The signature containment relation SigSub is defined as:

$$\mathsf{SigSub} \cdot F \cdot G = \forall\, X{:}\star.\, \forall\, Y{:}\star.\, \mathsf{Cast} \cdot X \cdot Y \to \mathsf{Cast} \cdot (F \cdot X) \cdot (G \cdot Y)$$

If there is a witness of $\mathsf{SigSub} \cdot F \cdot G$ for two type schemes F, G, we say that F *is contained in* G, or F is a *subsignature* of G.

Proposition 5.2. *For two type schemes $F, G : \star \to \star$ that both satisfy the predicate Mono, if F is contained in G then μF is included (via Cast) into μG.*

Proof. The proof is formalized in Cedille in the code repository associated with this paper. It comes as a direct consequence of the reuse combinator ifix2fix of Diehl et al. [9].

□

5.3 Generic Sufficient Condition For Constructor Subtyping

We now generalize our scheme for defining datatype signatures supporting flexible constructor subtyping so that we may instantiate these generic frameworks.

Definition 5.3 (Sig, the generic datatype signature). Given $A : \star$ (the labeling type) and a type family $B : \star \to A \to \star$ (the type family of constructor arguments indexed over labels $a : A$), we define the generic datatype signature Sig as

$$\mathsf{Sig} \cdot A \cdot B \quad = \quad \lambda\, R{:}\star.\, \Sigma\, a{:}A.\, B \cdot R\ a$$

where Σ is the derivable type constructor for dependent pairs (summarized in Figure 7). Here, the role of the abstracted type R is to stand in for recursive occurrences of the datatype $\mu(\mathsf{Sig} \cdot A \cdot B)$ in the constructor argument types.

The type family Sig over-generalizes the signatures of datatypes *CSNat* and *CSInt*: we do not assume that A has decidable equality, nor that B is formed from intersections of types whose first argument is an erased equational constraint on the given label. Restrictions on A and B would instead be enforced in the translation of syntax for datatype declarations supporting constructor subtyping to impredicative encodings. However, Sig provides additional flexibility for a surface-language implementation, as it permits each datatype signature to use a different labeling set (rather than the single type L we have used in our case studies).

To use the datatype $\mu(\mathsf{Sig} \cdot A \cdot B)$, we must be able to determine when the type scheme $\mathsf{Sig} \cdot A \cdot B$ is covariant. We have this when B is covariant in its type argument.

Lemma 5.4 (Covariance of Sig $\cdot A \cdot B$). *Assume $A : \star$ and $B : \star \to A \to \star$. If for all $a : A$ the type scheme $\lambda\, R : \star.\, B \cdot R\ a$ is Mono, then so is $\mathsf{Sig} \cdot A \cdot B$.*

Proof. Straightforward, as Σ is a positive type constructor. □

The main result of this section is a sufficient condition for signature containment for type schemes defined using Sig. This, in combination with Proposition. 5.2, gives in turn a sufficient condition for when datatypes whose signatures are defined using Sig are in the subtyping relation.

Theorem 5.5. *Assume labeling types $A_1, A_2 : \star$ and type families for constructor argument types $B_1 : \star \to A_1 \to \star$ and $B_2 : \star \to A_2 \to \star$ that are covariant in their respective type arguments. If*

- *A_1 is a subtype of A_2, witnessed by c*
- *and for all $a_1 : A_1$ and $R : \star$, we have $B_1 \cdot R\ a_1$ is a subtype of $B_2 \cdot R$ (cast -c a_1)*

Then $\mathsf{Sig} \cdot A_1 \cdot B_1$ and $\mathsf{Sig} \cdot A_2 \cdot B_2$ are in the signature containment relation SigSub (Def. 5.1)

Proof. Formalized in the Cedille code repository. □

We unpack the statement of Theorem 5.5. Our first condition is that the set of labels of type A_1 is contained within the set of labels of type A_2. The second condition requires that the constructor arguments types corresponding to the same label are in the subtyping relation when all recursive occurrences of both datatypes have been replaced by R. This means in particular that an implementation needs only to consider the assignment of labels and the types of the constructor arguments modulo those recursive occurrences.

5.4 Example: Naturals and Integers

We now return to our earlier motivating example of constructor subtyping for natural numbers and integers to demonstrate the use of the generic development. Unlike the earlier formulation in which the types of constructors were themselves packaged together in a type family, the generic framework of Firsov et al. [11] provides a single generic constructor in, and for the encoding we pack together just the constructor argument types.

A surface-language implementation using L to label all datatype constructors could elaborate the datatype declaration for *Nat* to:

$$\begin{aligned} \mathit{NatPack} \cdot R\ \ell &= (\{\ell \simeq \mathsf{lzero}\} \Rightarrow \mathsf{Unit}) \\ &\quad \cap (\{\ell \simeq \mathsf{lsucc}\} \Rightarrow R) \\ &\quad \cap (\neg\{\ell \simeq \mathsf{lzero}\} \cap \neg\{\ell \simeq \mathsf{lsucc}\} \Rightarrow \bot) \\ \mathit{NatSig} &= \mathsf{Sig} \cdot \mathsf{L} \cdot \mathit{NatPack} \\ \mathit{Nat} &= \mu \mathit{NatSig} \end{aligned}$$

where Unit is the singleton type (for constructors with no arguments) and $\bot$ is the empty type, and $\neg T = T \Rightarrow \bot$. Note that by choosing L for the labeling set, we require an explicit constraint that there are no other constructors. This could be avoided by instead using a subtype of L that only contains lzero and lsucc. Also, it is clear that *NatPack* is positive in its type argument.

With the generic encoding, an implementation would be able to define *now* constructors that may be reused for both the datatype *Nat* and all *future* declarations of datatypes that extend it. We begin with the polymorphic constructors for *NatPack*, which discharge impossible cases with the strong Principle of Explosion (Figure 5) to ensure the erasures of each component of the intersection are equal. Then, the constructors for the signature *NatSig* simply tuple these constructors with the appropriate label.

$$\begin{aligned} &\mathit{zeroP} : \forall\, N : \star.\ \mathit{NatPack}\ \mathsf{lzero} \cdot N \\ &\mathit{zeroP} = [\Lambda\, e.\ \mathsf{unit},\ [\Lambda\, e.\ \mathsf{explode}\ \text{-}(\delta - e)\ \beta\{\mathsf{unit}\}, \\ &\qquad \Lambda\, e.\ \mathsf{explode}\ \text{-}(e.1\ \text{-}\beta)\ \beta\{\mathsf{unit}\}]] \end{aligned}$$

$$\begin{aligned} &\mathit{succP} : \forall\, N : \star.\ N \to \mathit{NatPack}\ \mathsf{lsucc} \cdot N \\ &\mathit{succP}\ n = [\Lambda\, e.\ \mathsf{explode}\ \text{-}(\delta - e)\ \beta\{n\},\ [\Lambda\, e.\ n, \\ &\qquad \Lambda\, e.\ \mathsf{explode}\ \text{-}(e.2\ \text{-}\beta)\ \beta\{n\}]] \end{aligned}$$

$$\begin{aligned} \mathit{zeroS} &= (\mathsf{lzero}, \mathit{zeroP}) \\ \mathit{succS}\ n &= (\mathsf{lsucc}, \mathit{succP}\ n) \end{aligned}$$

For the extensible constructors of *Nat*, we take as module parameters a datatype signature F whose monotonicity is witnessed by *mono* and for which we have a proof $\mathit{sub} : \mathsf{SigSub} \cdot \mathit{NatSig} \cdot F$ that F contains *NatSig* (we expect an implementation to produce these). An immediate consequence of *sub* and the erasure of cast is the existence of an injection function $\mathit{ns2f} : \forall\, X : \star.\ \mathit{NatSig} \cdot X \to F \cdot X$ which erases to $\lambda\, x.\, x$. With *ns2f*, we can define the zero and successor constructors for the (arbitrary) supertype μF of *Nat*.

$$\begin{aligned} \mathit{zeroE} &= \mathsf{in}\ \text{-}\mathit{mono}\ (\mathit{ns2f}\ \mathit{zeroS}) \\ \mathit{succE}\ n &= \mathsf{in}\ \text{-}\mathit{mono}\ (\mathit{ns2f}\ (\mathit{succS}\ n)) \end{aligned}$$

The constructors *zero* and *succ* of *Nat* itself, then, arise from instantiating the module parameters with *NatSig* for the type scheme, the proof it is monotonic, and a lemma that every monotonic signature is a subsignature of itself.

Our declaration for *Int* given at the outset of this section explicitly requires that the labels for the constructors *izero* and *isucc* be the same as the labels for the constructors *zero* and *succ* of *Nat*. From this, an implementation could elaborate:

$$\begin{aligned} \mathit{IntPack} \cdot R\ \ell &= (\{\ell \simeq \mathsf{lzero}\} \Rightarrow \mathsf{Unit}) \\ &\quad \cap (\{\ell \simeq \mathsf{lsucc}\} \Rightarrow R) \\ &\quad \cap (\{\ell \simeq \mathsf{lipred}\} \Rightarrow R) \\ &\quad \cap (\neg\{\ell \simeq \mathsf{lzero}\} \cap \neg\{\ell \simeq \mathsf{lsucc}\} \cap \neg\{\ell \simeq \mathsf{lipred}\} \\ &\qquad \Rightarrow \bot) \\ \mathit{IntSig} &= \mathsf{Sig} \cdot \mathsf{L} \cdot \mathit{IntPack} \\ \mathit{Int} &= \mu \mathit{IntSig} \end{aligned}$$

where lipred is some new label. Again we see that *IntPack* is positive in its first argument.

It is at this point that an implementation must prove that, with the given identification of constructor labels, *NatSig* is a subsignature of *IntSig*. Once signature containment is established, by Theorem 5.2 we have that *Nat* is a subtype of *Int*. This also enables us to reuse the extensible constructors for *Nat* as constructors *izero* and *isucc* for *Int*, the purpose of which we will see in Section 5.5.

Proposition 5.6. *NatSig is a subsignature of IntSig.*

Proof. Formalized in the Cedille code repository. As this is a proof we expect an implementation to generate, we also give it in plain language to illustrate the reasoning that is to be automated.

By Theorem 5.5, it suffices to show an inclusion of the type $\mathit{NatPack} \cdot R\ \ell$ into $\mathit{IntPack} \cdot R\ \ell$ for all $\ell : \mathsf{L}$ and $R : \star$. This, in turn, is done by defining a function from the first type to the second type that behaves like the identity function.

So, assume we have some $x : \mathit{NatPack} \cdot R\ \ell$. To return something of type $\mathit{IntPack} \cdot R\ \ell$, we must give four terms that are all equal to x and each of whose types correspond to the types that are intersected in the definition of *IntPack*.

- $\{\ell \simeq \mathsf{lzero}\} \Rightarrow \mathsf{Unit}$:
  This is precisely the type of $x.1$, which erases to x.
- $\{\ell \simeq \mathsf{lsucc}\} \Rightarrow R$:
  This is precisely the type of $x.2.1$, which erases to x.
- $\{\ell \simeq \mathsf{lipred}\} \Rightarrow R$:
  We assume $\{\ell \simeq \mathsf{lipred}\}$, which implies that $\neg\{\ell \simeq \mathsf{lzero}\}$ and $\neg\{\ell \simeq \mathsf{lsucc}\}$. These two consequences can be defined such that they share the same erasure (see the code repository for this), meaning we have a proof of $\neg\{\ell \simeq \mathsf{lzero}\} \cap \neg\{\ell \simeq \mathsf{lsucc}\}$. From this and $x.2.2$ we have a proof of $\bot$, and since the argument of $x.2.2$ is erased, the whole proof erases to x.
- $\neg\{\ell \simeq \mathsf{lzero}\} \cap \neg\{\ell \simeq \mathsf{lsucc}\} \cap \neg\{\ell \simeq \mathsf{lipred}\} \Rightarrow \bot$:
  From a proof e that ℓ is neither lzero, lsucc, nor lipred, we have that $[e.1, e.2.1]$ has type $\neg\{\ell \simeq \mathsf{lzero}\} \cap \neg\{\ell \simeq \mathsf{lsucc}\}$. From this and $x.2.2$ we have a proof of $\bot$, and since the argument to $x.2.2$ is erased, the whole proof erases to x.

□

To complete the definition of the datatype *Int*, we need to define the constructor *ipred* and can reuse *zeroE* for *izero* and *succE* for *isucc*. The definition of *ipred* follows the same format as for the two constructors of *Nat*: first define the polymorphic constructors *ipredP* and *ipredS* for the signature *IntSig*, then the extensible constructor *ipredE* from which *ipred* follows as a special case.

$$\frac{t : NatSig \cdot N \quad t_1 : T \quad t_2 : N \to T}{caseNatSig\ t\ t_1\ t_2 : T}$$

$$\begin{array}{rcl} |caseNatSig\ zeroS\ t_1\ t_2| & = & |t_1| \\ |caseNatSig\ (succS\ t)\ t_1\ t_2| & = & |t_2\ t| \end{array}$$

Figure 8: Case-distinction principle for *NatSig*

5.5 Example: Extending Addition for Naturals

Using the extensible constructors for *Nat*, we can define extensible *functions* also. The idea follows the *datatypes á la carte* approach of Swierstra [28], though without the benefit of Haskell's type class instance resolution to automatically insert these coercions. As an example, we show how one could give a definition for addition over natural numbers that can be reused for supertypes.

We first illustrate the idea with pseudocode. In a higher-level surface language, we would like to write something like the following for an extensible definition of addition.

$$\begin{array}{l} add : \forall\, D \geqslant Nat.\, Nat \to D \to D \\ add\ zero\ n = n \\ add\ (suc\ m')\ n = D.succ\ (add\ m'\ n) \end{array}$$

That is, we wish to quantify over all supertypes D of *Nat* and compute a sum of type D, where in the successor case we use the constructor of D corresponding to *succ*. The function *nadd* for addition over natural numbers would then be recovered as an instance: $nadd = add \cdot Nat$.

While the second argument of *add* has the supertype D, here we propose to require that the first argument has type *Nat*. This is because the first argument is the one over which *add* is inductively defined, and we would like the implementation's coverage checker to confirm *add* is total. Therefore, we are really considering *two* forms of reuse: one using bounded quantification, and one in which we extend the list of clauses of a recursive definition. For example, to extend *add* to *Int*, we would like to write the following.

$$\begin{array}{l} iadd : \forall\, D \geqslant Int.\, Int \to D \to D \\ iadd\ = \text{extend } add \text{ where} \\ \qquad iadd\ (ipred\ m')\ n = D.ipred\ (iadd\ m\ n) \end{array}$$

We now demonstrate how these features for function reuse can be realized by the generic encoding.

Assume we have a type scheme $F : \star \to \star$ that is monotonic and a supersignature of *NatSig*. The shape of our recursive definition for *add* is given by *addAlg*.

$$\begin{array}{l} addAlg : \mathsf{Alg} \cdot NatSig \cdot (\mu F \to \mu F) \\ addAlg\ add\ m\ n = caseNatSig\ m\ n \\ \qquad \lambda\, m'.\, succE\ (add\ m'\ n) \end{array}$$

The bound variable *add* of type $R \to \mu F \to \mu F$ is the handle for recursive calls, R is (implicitly) a type argument to *addAlg* (see Alg in Figure 6), and *caseNatSig* is the case distinction principle for *NatSig*, whose typing and computation rule are summarized in Figure 8. The supertype's constructor for successor, written *D.succ* in the pseudocode listing, is implemented with the extensible constructor *succE*. Ordinary addition for natural numbers is obtained by instantiating F with *NatSig* and using fold (Figure 6):

$$nadd = \text{fold } addAlg$$

To extend a function recursively defined over *Nat* to some supertype, the implementation must determine the set of constructor labels not shared with *Nat* and require the user to provide clauses for the constructors associated to those labels. For *Int*, we are only missing a clause for *ipred*. Function *n2ia* below shows how this extension, invoked by the "extend" keyword in the pseudocode listing, is achieved with the encoding.

$$\begin{array}{l} n2iA : \forall\, X : \star.\, \mathsf{Alg} \cdot NatSig \cdot X \to \\ \qquad \mathsf{Alg} \cdot (IntPack\ \mathsf{lipred}) \cdot X \to \mathsf{Alg} \cdot IntSig \cdot X \\ n2iA\ a\ p\ rec\ i = caseIntSig\ i\ (a\ rec\ zeroS)\ (\lambda\, i'.\, a\ rec\ (succS\ i')) \\ \qquad (\lambda\, i'.\, p\ rec\ (ipredP\ i')) \end{array}$$

The first argument a of *n2iA* is the shape of the (arbitrary) function for naturals we wish to extend, and the second argument p explains how to extend this function for the *ipred* constructor. In the definition, we use the case distinction principle *caseIntSig* for *IntSig* (with similar typing and computation rules as that of *caseNatSig*) on the given $i : IntSig \cdot Int$. In the case that it is zero or constructed with successor, we dispatch to the given function for naturals; if it is constructed using predecessor, we invoke the extension p given just for this case.

Note that while constructor subtyping is zero-cost, function extension as provided by *n2ia* is not: even if we avoid rebuilding subdata with constructors after performing case distinction, as occurs in *n2iA*, in order to downcast some $IntPack\ \ell \cdot R$ to the type $NatPack\ \ell \cdot R$ we must dynamically check the label ℓ to ensure that $\ell = \mathsf{lzero}$ or $\ell = \mathsf{lsucc}$, introducing runtime overhead.

Finishing the example, for the definition of *iadd* we have:

$$\begin{array}{l} iaddAlg = n2ia\ addAlg \\ \qquad (\lambda\, iadd.\, \lambda\, m'.\, \lambda\, n.\, ipredE\ (iadd\ (m'.2.2.1\text{-}\beta)\ n)) \end{array}$$

$$iadd = \text{fold } iaddAlg$$

In *iaddAlg*, m' corresponds to the value of the same name bound by the pattern *ipred* m' in the pseudocode definition of *iadd*. Here m' has type *IntPack* lipred, so we use intersection projection and the proof β (here checked against type $\{\mathsf{lipred} \simeq \mathsf{lipred}\}$) to retype it. Finally, *iadd* itself is implemented using fold and *iaddAlg*.

6 RELATED WORK

As previously mentioned, calculi with constructor subtyping and other desirable properties have been developed and explored by Barthe et al. [3, 4]. However, there are many other approaches to subtyping that could enable similar features such as coercive subtyping [19, 30], algebraic subtyping [10], and semantic subtyping [5, 12, 13] to name a few. The encoding we present in this paper corresponds to semantic subtyping, except that the subtyping relation is internally derived. To our knowledge, Constructor subtyping of indexed types has not been explored before in any subtyping framework. The syntactic subtyping approaches (Barthe's calculi,

coercive subtyping, algebraic subtyping, etc) place a heavier burden on the meta-theory of the language and limit the potential flexibility of the subtyping relation at the point of definition. In contrast, our encoding shifts the burden to internal semantic subtyping which requires proofs of the subtyping relationship (as opposed to it being a syntactic check or inference).

Research in Object Oriented Programming (OOP) has extensively explored the idea of method overloading [6, 7, 25]. Indeed, method overloading is a common feature of almost all industry OOP languages. A closely related notion, inheritance, enables similar features to the structural subtyping discussed in functional languages. It is not clear that our method of encoding could be used to model inheritance. Moreover, in this work we present a method of *function extension* similar to data types à la carte [28] which is not the same as function overloading via mechanisms like type classes present in functional languages like Haskell.

Ornaments have been used for proof reuse of inductive datatypes in Coq, although they require the same inductive structure [24]. Proof and program reuse was not the principal goal of this work, however ornaments yield linear time reuse for inductive data types defined via the ornament mechanism. While zero-cost constructor subtyping yields constant time reuse for proofs and functions (though not zero-cost function extension, as seen in Section 5.5).

7 CONCLUSIONS AND FUTURE WORK

In this paper we have devised a way to derive inductive datatypes that support constructor subtyping where the subtyping relation is type inclusion. At its core, the encoding works by deriving a form of disjoint union of constructors over some indexing label type. This encoding continues a line of work in Cedille of constructing inductive data via lambda encodings that aims to demonstrate that its theory, CDLE, can serve as a kernel language for interactive theorem provers with expressive surface-language features. Our generic result in particular makes progress towards realizing an implementation of zero-cost constructor subtyping in Cedille. Looking forward, we believe that the method of function extension described in Section 5 can be further improved. Indeed, the labeling type used in the encoding was required to have decidable equality, but in principle many additional properties can be imposed (such as a total order) which could reduce the overhead of label comparisons.

REFERENCES

[1] Andreas Abel, Ralph Matthes, and Tarmo Uustalu. 2005. Iteration and coiteration schemes for higher-order and nested datatypes. *Theor. Comput. Sci.* 333, 1-2 (2005), 3–66. https://doi.org/10.1016/j.tcs.2004.10.017

[2] Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and E. Moran. 2006. Innovations in computational type theory using Nuprl. *J. Applied Logic* 4, 4 (2006), 428–469. https://doi.org/10.1016/j.jal.2005.10.005

[3] Gilles Barthe and Maria João Frade. 1999. Constructor subtyping. In *European Symposium on Programming*. Springer, 109–127.

[4] Gilles Barthe and Femke Van Raamsdonk. 2000. Constructor subtyping in the calculus of inductive constructions. In *International Conference on Foundations of Software Science and Computation Structures*. Springer, 17–34.

[5] Giuseppe Castagna and Alain Frisch. 2005. A gentle introduction to semantic subtyping. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*. 198–199.

[6] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. 1995. A calculus for overloaded functions with subtyping. *Information and Computation* 117, 1 (1995), 115–135.

[7] Daniel KC Chan and Philip W Trinder. 1994. An object-oriented data model supporting multi-methods, multiple inheritance, and static type checking: A specification in Z. In *Z User Workshop, Cambridge 1994*. Springer, 297–315.

[8] Thierry Coquand. 1992. Pattern matching with dependent types. In *Informal proceedings of Logical Frameworks*, Vol. 92. Citeseer, 66–79.

[9] Larry Diehl, Denis Firsov, and Aaron Stump. 2018. Generic zero-cost reuse for dependent types. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–30.

[10] Stephen Dolan. 2017. *Algebraic subtyping*. BCS, The Chartered Institute for IT.

[11] Denis Firsov, Richard Blair, and Aaron Stump. 2018. Efficient Mendler-Style Lambda-Encodings in Cedille. In *International Conference on Interactive Theorem Proving*. Springer, 235–252.

[12] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2002. Semantic subtyping. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 137–146.

[13] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM (JACM)* 55, 4 (2008), 1–64.

[14] Herman Geuvers. 2001. Induction Is Not Derivable in Second Order Dependent Type Theory. In *International Conference on Typed Lambda Calculi and Applications*, Samson Abramsky (Ed.). Springer, Berlin, Heidelberg, 166–181.

[15] Ralf Hinze. 2002. Polytypic values possess polykinded types. *Sci. Comput. Program.* 43, 2-3 (2002), 129–159. https://doi.org/10.1016/S0167-6423(02)00025-4

[16] Christopher Jenkins and Aaron Stump. 2020. Monotone recursive types and recursive data representations in Cedille. arXiv:2001.02828 [cs.PL] Under consideration for publication in J. Mathematically Structured Computer Science.

[17] Alexei Kopylov. 2003. Dependent Intersection: A New Way of Defining Records in Type Theory. In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science (LICS '03)*. IEEE Computer Society, Washington, DC, USA, 86–.

[18] Joachim Lambek. 1968. A Fixpoint Theorem for Complete Categories. *Mathematische Zeitschrift* 103, 2 (1968), 151–161. https://doi.org/10.1007/bf01110627

[19] Zhaohui Luo. 1999. Coercive subtyping. *Journal of Logic and Computation* 9, 1 (1999), 105–130.

[20] Ralph Matthes. 1998. Monotone Fixed-Point Types and Strong Normalization. In *Computer Science Logic, 12th International Workshop, CSL '98, Annual Conference of the EACSL, Brno, Czech Republic, August 24-28, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1584)*, Georg Gottlob, Etienne Grandjean, and Katrin Seyr (Eds.). Springer, 298–312. https://doi.org/10.1007/10703163_20

[21] N. P. Mendler. 1987. Recursive Types and Type Constraints in Second-Order Lambda Calculus. In *Proceedings of the Symposium on Logic in Computer Science ((LICS '87))*. IEEE Computer Society, Los Alamitos, CA, 30–36.

[22] Nax Paul Mendler. 1991. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic* 51, 1 (1991), 159 – 172. https://doi.org/10.1016/0168-0072(91)90069-X

[23] Alexandre Miquel. 2001. The Implicit Calculus of Constructions: Extending Pure Type Systems with an Intersection Type Binder and Subtyping. In *Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications* (Kraków, Poland) *(TLCA'01)*. Springer-Verlag, Berlin, Heidelberg, 344–359.

[24] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2019. Ornaments for proof reuse in Coq. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[25] Francois Rouaix. 1989. Safe run-time overloading. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 355–366.

[26] Aaron Stump. 2017. The calculus of dependent lambda eliminations. *Journal of Functional Programming* 27 (2017), e14.

[27] Aaron Stump. 2018. From realizability to induction via dependent intersection. *Ann. Pure Appl. Logic* 169, 7 (2018), 637–655. https://doi.org/10.1016/j.apal.2018.03.002

[28] Wouter Swierstra. 2008. Data types à la carte. *J. Funct. Program.* 18, 4 (2008), 423–436. https://doi.org/10.1017/S0956796808006758

[29] Tarmo Uustalu and Varmo Vene. 1999. Mendler-style Inductive Types, Categorically. *Nordic Journal of Computing* 6, 3 (Sep 1999), 343–361. http://dl.acm.org/citation.cfm?id=774455.774462

[30] Tao Xue. 2013. *Theory and implementation of coercive subtyping*. Ph.D. Dissertation. Royal Holloway, University of London, UK.