Convenient Algebraic Programming With Coercive Subtyping

Henry Blanchette[1][0000−0002−9415−0944] and Aaron Stump[2][0000−0002−9720−0003]

[1] University of Maryland, College Park MD 20742, USA
blancheh@umd.edu
[2] Boston College, Chestnut Hill MA 02467, USA
aaron.stump@bc.edu

Abstract. This paper presents a system for more convenient algebraic programming, where datatypes are defined as least fixed points of covariant signature functors, and recursive functions are defined as homomorphisms from initial algebras. This approach requires rolling and unrolling fixed points, and conversion of algebras to functions that recurse over data. To alleviate the burden of these extra operations, in this paper we propose to insert them automatically, using coercive subtyping. The type checker generates subtyping constraints, which are then solved with the help of a custom logic programming engine called Chronolog. The subtyping axioms are given to Chronolog as rules, which uses them to solve the subtyping constraints. To avoid infinite applications of rules for unrolling signature functors, Chronolog provides a mechanism for suspending and resuming certain forms of constraints. The subtyping axioms presented to Chronolog are an equivalent algorithmic version of the usual declarative rules for algebraic programming.

Keywords: termination, algebraic programming, coercive subtyping

1 Introduction

The discipline of strong functional programming statically requires that functions terminate on all inputs. One way to achieve this is through an algebraic approach: datatypes are least fixed points μF of signature functors F, and recursive functions are least fixed points $(\!|a|\!)$, called catamorphisms, of algebras a. The functor F describes one layer of the data, and μF then describes data consisting of any finite number of layers. We may unroll μF to $F\ \mu F$, and roll it back again. Algebras interpret the operations declared for one layer of the data, and catamorphisms apply algebras across all the layers. There is no other mechanism for recursion or looping in the language, leading to a strong guarantee: all functions written in this style terminate.

One burden is the need to include calls to roll and unroll functors, and to $(\!|\cdot|\!)$, in code. This adds clutter, compared to regular functional programming. In this paper, we address this problem by automatically inserting those coercions wherever they are needed in code. The resulting code looks indistinguishable

from regular pure-functional code, and yet falls within the algebraic style, thus assuring termination. To insert these calls automatically, we turn to *coercive subtyping*. The type checker generates subtyping constraints that need to hold, in order for the term to be well-typed. The constraints are processed by a constraint solver, generating coercions that are then inserted at the places where the subtyping constraints were generated.

To implement this, we make use of a custom logic-programming engine called Chronolog. The subtyping axioms are given to Chronolog as rules. Chronolog then uses standard SLD resolution ([2, 10]) to search for proofs of the constraints, instantiating type variables during proof search via unification. The rules for rolling and unrolling signatures would, if handled naively, lead to diverging proof search. Instead, Chronolog provides a mechanism to indicate that certain problematic constraints should be suspended. These suspended constraints may be resumed if they are instantiated during solving of other constraints. If not, an outer loop of our algorithm handles them specially.

Standard subtyping rules generally are not suitable for logic programming, due to the presence of a transitivity rule, which nondeterministically guesses B with $A <: B$ and $B <: C$, to solve $A <: C$. We identify (Section 5) an algorithmic set of subtyping rules, without transitivity, and prove it equivalent, in Agda, to the declarative rules. This both ensures we can use logic programming (with our algorithmic rules) to solve these subtyping constraints, and also assures us that the algorithmic rules are indeed correct. To summarize our contributions:

- We propose a new approach to total algebraic programming, where calls for rolling and unrolling signature functors and turning algebras into catamorphisms are inferred by coercive subtyping.
- We realize this approach with constraint-generating typing, and a constraint solver based on logic programming, modified to avoid infinite unrolling of functors. We prove termination of our algorithm.
- We propose an algorithmic subtyping system, and prove it equivalent to the declarative one.

The implementation and proofs, respectively, are available at:
https://github.com/rybla/algebraic-programming-with-coercive-subtyping
https://github.com/astump/chronolog-algebraic-paper

2 Syntax

Figure 1 presents the syntax for the language $\mathcal{L}_{(\!|\cdot|\!)}$, through which we study coercion inference for algebraic programming. We use notation $\overline{e \oplus}$ where $\oplus$ is a syntactic operator, to indicate a $\oplus$-separated list of expressions e. Datatypes are based on signature functors F, which we take to be polynomial functors for simplicity. Least fixed points of signature functors, referenced by datatype name D, are denoted μD. We have usual function types $A \to B$. Types $D \Rightarrow B$ are the types for algebras over μD computing a value of type B, the carrier of the algebra. Recursion type variables R_f are introduced in the bodies of algebras,

where f is the name of the function introduced by the respective algebra. While type variables (which are unrelated to recursion type variables) appear during type checking, as detailed in Section 6, they are not allowed in the surface syntax for types. In this way, all types that appear in programs are ground types.

Terms include the usual constructs of simply typed λ-calculus. There are also constructor applications $k(\overline{a,})$, and simple pattern-matching with notation $\gamma a.\{\overline{k(\overline{x,}) \mapsto b\ |}\}$, where the value of a is matched against constructor patterns $k(\overline{x,})$. If such a pattern matches, then the term reduces to the body b for that case. For totality, all constructors for a's datatype must be covered. To apply a coercion c to term a, we write $c - a$. Coercions, which include rolling, unrolling, and catamorphizing, will be explained with their typing rules in Section 6.

Algebras are denoted $\omega f(x).b$, which we adopt in so-called Mendler style [16]. Let us briefly review these. Standard F-algebras are typed as

$$a : F\ X \to X$$

where X is the carrier of the algebra; that is, the type of the values computed by the algebra. The algebra a takes input of type $F\ X$. This input looks like a value of the datatype, but with subdata replaced by their recursively computed values of type X. Given such an algebra, its catamorphism $(\!|a|\!)$ is typed as

$$(\!|a|\!) : \mu F \to X$$

With Mendler-style algebras, we have a different typing for a:

$$a : (R_f \to X) \to F\ R_f \to X$$

Here, a takes in two inputs. The first is a function f from R_f (the recursion type variable) to X. The second input is a value x of type $F\ R_f$. This is similar to the input for an F-algebra, but with R_f in place of the carrier. The recursion type variables are abstract: no other information about them is available to algebras.

Mendler algebras are closer to usual programming practice, because the algebra may pattern-match on x to reveal subdata of type R_f, and then apply f to those to recurse. This is syntactically similar to making a recursive call in a standard functional programming language.

A datatype is defined by a tuple $\langle D, F\rangle$ where D is its name and F is its signature functor. This approach was proposed and developed in early works like [8, 7]. For examples presented in this paper, we define the following set of datatypes.

$$\begin{aligned}\mathscr{D} := \{&\langle \mathit{Nat}, \lambda X.\mathit{Zero} + \mathit{Suc}(X)\rangle,\\ &\langle \mathit{NatList}, \lambda X.\mathit{Nil} + \mathit{Cons}(\mu \mathit{Nat} \times X)\rangle\}\end{aligned}$$

3 Motivating examples

Figure 2 shows code with explicit coercions, for two functions. The `sub` function subtracts two natural numbers, and `subs` proceeds through a two lists of numbers, subtracting corresponding elements. So subtracting the list (using Haskell

$$
\begin{array}{lrcl}
\text{(Datatype names)} & D & \in & \{\mathit{Nat}, \mathit{NatList}, \dots\} \\
\text{(Constructor names)} & k & \in & \{\mathit{Zero}, \mathit{Suc}, \mathit{Nil}, \mathit{Cons}, \dots\} \\
\text{(Term variables)} & x & \in & \textbf{TermVars} \\
\text{(Recursion type variables)} & R & ::= & R_f \\
\text{(Type variables)} & X, Y & \in & \textbf{TypeVars} \\
\text{(Contexts)} & \Gamma & ::= & \cdot \mid \Gamma, (x : A) \mid \Gamma, (R_f : *) \\
\text{(Datatype signature functor)} & F & ::= & \lambda X.\overline{k(\overline{P_X\ \times})\ +} \\
\text{(Constructor parameter types)} & P_X & ::= & X \mid \mu D \\
\text{(Types)} & A, B & ::= & X \mid R \mid \mu D \mid D\ A \mid A \to B \mid D \Rightarrow B \\
\text{(Terms)} & a, b, f & ::= & x \mid k(\overline{a,}) \mid \lambda(x : A).b \mid (f\ a) \mid \omega f(x).b \mid \\
 & & & \gamma a.\{\overline{k(\overline{x,}) \mapsto b\ \mid}\} \mid \mathit{let}\ x : A = a\ \mathit{in}\ b \mid c - a \\
\text{(Coercions)} & c, c_1, c_2 & ::= & \mathit{cata}\ c_1\ c_2 \mid \mathit{roll}\ c \mid \mathit{unroll}\ c \mid \mathit{cov}[D]\ c \mid \\
 & & & \mathit{id}[\mu D] \mid \mathit{arr}\ c_1\ c_2 \mid \mathit{alg}[D]\ c \mid \mathit{id}[R]
\end{array}
$$

Fig. 1: Syntax of $\mathcal{L}_{(\!|\cdot|\!)}$

syntax) $[1, 2, 3]$ from $[8, 9, 10]$ produces $[7, 7, 7]$, for example. Let us see how the different kinds of coercions are used. The `sub` function takes in the minuend `m` and returns an algebra, which can be used to recurse through the subtrahend `n`. The code begins with ω `sub'(n)`, which introduces the name `sub'` for recursive calls, and the input value `n`. The code then matches on this `n` with γ, returning `m` in the `Zero` case. In the `Suc(n')` case, we recursively subtract `n'` (from `m`). Since our algebras are Mendler-style, the type of `sub'` is `R_f` → μ `Nat`, where `R_f` is the abstract type introduced by the algebra. The input `n` has type `Nat R_f`, and so its subdatum `n'` has type `R_f`. So the call `sub' n'` is well-typed. To pattern-match on the output of this call requires unrolling μ `Nat` with `unroll id`, written `unroll id - sub' n'`. The `id` is the identity coercion, indicating that the subtyping is only happening at the top level of the type. In the `Zero` case of this match on unrolled `sub' n'`, the code returns `Zero`. Constructors' return types are applications of signature functors. Here, `Zero` can be given type `Nat (`μ `Nat)`, which is then coerced to μ `Nat` by `roll`, written `roll id - Zero`.

The function `subs` does similar rolling and unrolling, but also invokes subtraction (`sub`). Since `sub` returns an algebra, the code uses `cata` to coerce that to the function which accepts the subtrahend, written `cata id id - sub h1 h2`, where the the coercion operator `-` binds more tightly than application (i.e. the expression is parsed equivalently as `(cata id id - sub) h1 h2`). Note that `subs` returns a list as long as the shorter of its inputs, similar to `zip` in Haskell.

Thanks to the restrictions imposed by Mendler-style algebras, these functions have the desirable property that they are guaranteed to terminate on all inputs.

```
sub : μ Nat → Nat ⇒ μ Nat =
  λ m . ω sub'(n) . γ n .
    { Zero → m
    | Suc(n') → γ unroll id - sub' n' .
        { Zero → roll id - Zero
        | Suc(m') → m' } }

subs : NatList ⇒ μ NatList → μ NatList =
  ω subs'(l1) . λ l2 . γ l1 .
    { Nil → Nil
    | Cons(h1, t1) → γ unroll id - l2 .
        { Nil → roll id - Nil
        | Cons(h2, t2) →
          (roll id) -
            (Cons(cata id id - sub h1 h2, subs' t1 t2)) } }
```

Fig. 2: Algebraic implementation of `sub` and `subs`, with explicit coercions.

Nonterminating recursive calls are disallowed just by typing. Nevertheless, we expect the reader will agree that the code is rather burdened with coercions. This raises the question we address in this paper: can we support algebraic programming like this with lighter syntax? We return with our positive answer for these examples in Figure 8 below.

4 Typing for $\mathcal{L}_{(\!|\cdot|\!)}$

The type system of $\mathcal{L}_{(\!|\cdot|\!)}$ conservatively extends that of the simply-typed λ-calculus with specialized constructs for constructing and deconstructing datatypes, algebraic recursion, and explicit coercion. The typing rules for $\mathcal{L}_{(\!|\cdot|\!)}$ are presented in Figure 3.

The $\gamma a.\{\dots\}$ construct for pattern matching is typed by the Gamma rule. This rule resolves the signature functor associated with the datatype of the scrutinee. Because the signature functor abstracts over a single type parameter, Gamma instantiates it with the type argument C, satisfying the premise $\Gamma \vdash a : D\ C$.

The $\omega f(x).b$ construct for introducing an algebra is typed by the Omega rule, which accounts for the introduction of a fresh abstract type R_f into the local typing context of the body b. This type variable R_f is linked to the specific syntactic binder, a dependency captured by its index f. Because $\mathcal{L}_{(\!|\cdot|\!)}$ purposely omits general polymorphism and type inference, this mechanism serves as a highly restricted form of scoped type abstraction tailored explicitly for modeling Mendler-style algebras. The generative nature of the recursion type variable R_f isolates it from all other types in the context. Analogous to the role of relational parametricity in standard Mendler encodings, this opacity strongly guarantees that the invocation f can only ever be applied to sub-data of x.

$$\textsc{Var}\ \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Con}\ \frac{\langle D, \lambda X. \cdots + k(\overline{A_i\ \times}) + \cdots \rangle \in \mathscr{D} \qquad \overline{\Gamma \vdash a_i : A_i[X \mapsto C]}}{\Gamma \vdash k(\overline{a_i,}) : D\ C}$$

$$\textsc{Lam}\ \frac{\Gamma, (x : A) \vdash b : B}{\Gamma \vdash \lambda(x : A).b : A \to B} \qquad \textsc{App}\ \frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B}$$

$$\textsc{Omega}\ \frac{\Gamma, (R_f : *), (f : R_f \to A), (x : D\ R_f) \vdash b : B}{\Gamma \vdash \omega f(x).b : D \Rightarrow B}$$

$$\textsc{Gamma}\ \frac{\langle D, \lambda X. k_i(\overline{A_{ij}\ \times})\ + \rangle \in \mathscr{D} \qquad \Gamma \vdash a : D\ C \qquad \overline{\Gamma, \overline{x_{ij} : A_{ij}[X \mapsto C]}, \vdash b_i : B}}{\Gamma \vdash \gamma a. \{\overline{k_i(\overline{x_{ij},}) \mapsto b_i\ |}\} : B}$$

$$\textsc{Let}\ \frac{\Gamma, (x : A) \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathit{let}\ x : A = a\ \mathit{in}\ b : B} \qquad \textsc{Coerce}\ \frac{\Gamma \vdash c : A <:: B \qquad \Gamma \vdash a : A}{\Gamma \vdash c - a : B}$$

Fig. 3: Typing rules for $\mathcal{L}_{(\!|\cdot|\!)}$

The explicit coercion construct, $c - a$, is typed by the Coerce rule. This rule relies upon a coercion typing judgment of the form $\Gamma \vdash c : A <:: B$. To explain this, we must first consider the underlying subtyping relation.

5 Subtyping

We begin with the natural declarative subtyping rules. These are not algorithmic when read from conclusions to premises: they cannot be used directly as a logic programd. This is due to the Trans rule, which introduces a new metavariable in its premises that does not appear in in its conclusion, and so can be applied ad infinitum. So we next propose an algorithmic version, where all metavariables in the premises of the rule are already present in the conclusion, and the sizes of the constraints decrease. This ensures that SLD resolution will not diverge on ground constraints. We prove that the two sets of rules are equivalent. Then we can formulate the coercion typing rules.

5.1 Declarative subtyping

Figure 4 shows the rules for declarative subtyping $A <: B$. We have reflexivity rule Refl for datatypes. Arr is the usual subtyping rule for function types, which is contravariant in the domain and covariant in the codomain. Cov expresses that the signature functor is indeed a (covariant) functor. Alg expresses

$$\textsc{Refl}\ \frac{}{\mu D <: \mu D} \qquad \textsc{Arr}\ \frac{A' <: A \quad B <: B'}{A \to B <: A' \to B'} \qquad \textsc{Cov}\ \frac{A <: A'}{D\ A <: D\ A'} \qquad \textsc{Alg}\ \frac{A <: A'}{D \Rightarrow A <: D \Rightarrow A'}$$

$$\textsc{Roll}\ \frac{}{D\ \mu D <: \mu D} \qquad \textsc{Unroll}\ \frac{}{\mu D <: D\ \mu D} \qquad \textsc{Cata}\ \frac{}{D \Rightarrow A <: \mu D \to A} \qquad \textsc{Trans}\ \frac{A <: A' \quad A' < A''}{A <: A''}$$

Fig. 4: Declarative subtyping rules.

$$\textsc{Refl}\ \frac{}{\mu D <:: \mu D} \qquad \textsc{Arr}\ \frac{A' <:: A \quad B <:: B'}{A \to B <:: A' \to B'} \qquad \textsc{Cov}\ \frac{A <:: A'}{D\ A <:: D\ A'} \qquad \textsc{Alg}\ \frac{A <:: A'}{D \Rightarrow A <:: D \Rightarrow A'}$$

$$\textsc{Roll}\ \frac{A <:: \mu D}{D\ A <:: \mu D} \qquad \textsc{Unroll}\ \frac{\mu D <:: A}{\mu D <:: D\ A} \qquad \textsc{Cata}\ \frac{B <:: \mu D \quad A <:: C}{D \Rightarrow A <:: B \to C}$$

Fig. 5: Algorithmic subtyping rules.

that algebra types $D \Rightarrow A$ are covariant in their carriers A. Roll and Unroll together express that μF is equivalent to $F\ \mu F$. Cata expresses that an algebra can be coerced to a function, namely the algebra's catamorphism. Finally, there is the Trans rule, completing the definition of subtyping as a preorder.

5.2 Algorithmic subtyping

Figure 5 presents the rules for algorithmic subtyping $A <:: B$. We use similar names as for the declarative rules, but critically, the algorithmically problematic Trans rule is not present. A consequence of this is that now several rules that did not have premises in the declarative formulation, here do. For example, compare the declarative (on the left) and algorithmic Cata rules:

$$\frac{}{D \Rightarrow A <: \mu D \to A} \qquad \frac{B <:: \mu D \quad A <:: C}{D \Rightarrow A <:: B \to C}$$

The algorithmic rule needs to take into account subtyping relationships between the components in the conclusion, because there is no rule for transitivity, which could allow those relationships to be taken into account after this rule is applied. In effect, transitivity needs to be built into all the other rules. We see a similar change from the declarative to the algorithmic versions of the Roll and Unroll rules, for rolling and unrolling the signature functor.

Lemma 1 (Decrease). *For all rules of Figure 5, the sizes of the constraints in the premises are less than the size of the constraint in the conclusion.*

Theorem 1 (Decidability). *It is decidable whether or not two types are in the algorithmic subtyping relation.*

Proof. Bottom-up application of the rules to ground types is guaranteed to terminate, by Lemma 1. Therefore, proof search may be used as a decision procedure for algorithmic subtyping of ground types.

5.3 Equivalence of declarative and algorithmic subtyping

While Theorem 1 ensures that we can test ground subtyping constraints using proof search, we must confirm that the algorithmic rules are indeed a correct version of the declarative ones. For this, we prove the following theorems. All theorems in this subsection have been formalized and proven in Agda.

Theorem 2 (Soundness). $A <:: B$ *implies* $A <: B$.

Theorem 3 (Completeness). $A <: B$ *implies* $A <:: B$.

The sets of rules are quite similar, so these proofs are not onerous. They do depend on the following lemmas:

Lemma 2 (roll/unroll declarative/algorithmic).

1. *If* $A <: \mu D$, *then* $D\ A <: \mu D$.
2. *If* $\mu D <: A$, *then* $\mu D <: D\ A$.
3. $D\ \mu D <:: \mu D$
4. $\mu D <:: D\ \mu D$

Lemma 3 (preorder algorithmic).

1. $A <:: A$.
2. *If* $A <:: B$ *and* $B <:: C$ *then* $A <:: C$.

As an aside: with the declarative subtyping rules, there are multiple derivations for any derivable subtyping statement. This is due to the presence of transitivity, and a derivable reflexivity rule. But for algorithmic subtyping, the situation is much more constrained:

Theorem 4 (Uniqueness). *There is at most one derivation of* $A <:: B$.

5.4 Typing coercions

Having established the formalization of subtyping in $\mathcal{L}_{(\!|\cdot|\!)-}$, we present the coercion typing system for $\mathcal{L}_{(\!|\cdot|\!)}$ in Figure 6. These rules exhibit an exact structural correspondence with the algorithmic subtyping relation defined in Figure 5. This isomorphism enables a systematic, constructive translation from any valid subtyping derivation $A <:: B$ to a coercion typing derivation $c : A <:: B$. Consequently, this correspondence forms the foundation of our elaboration semantics,

$$\textsc{ReflDataCoe}\ \frac{}{\Gamma \vdash id[\mu D] : \mu D <:: \mu D} \qquad \textsc{RecVarCoe}\ \frac{}{\Gamma \vdash id[R_f] : R_f <:: R_f}$$

$$\textsc{ArrCoe}\ \frac{\Gamma \vdash c_1 : A' <:: A \qquad \Gamma \vdash c_2 : B <:: B'}{\Gamma \vdash arr\ c_1\ c_2 : A \to B <:: A' \to B'} \qquad \textsc{CovCoe}\ \frac{\Gamma \vdash c : A <:: A'}{\Gamma \vdash cov[D]\ c : D\ A <:: D\ A'}$$

$$\textsc{AlgCoe}\ \frac{\Gamma \vdash c : A <:: A'}{\Gamma \vdash alg[D]\ c : D \Rightarrow A <:: D \Rightarrow A'} \qquad \textsc{RollCoe}\ \frac{\Gamma \vdash c : A <:: \mu D}{\Gamma \vdash roll\ c : D\ A <:: \mu D}$$

$$\textsc{UnrollCoe}\ \frac{\Gamma \vdash c : \mu D <:: A}{\Gamma \vdash unroll\ c : D\ A <:: \mu D} \qquad \textsc{CataCoe}\ \frac{\Gamma \vdash c_1 : B <:: D \qquad \Gamma \vdash c_2 : A <:: C}{\Gamma \vdash cata\ c_1\ c_2 : D \Rightarrow A <:: B \to C}$$

Fig. 6: Coercion typing rules.

allowing us to infer and insert the explicit coercions necessary to translate a well-typed $\mathcal{L}_{(\!|\cdot|\!)-}$ term into a valid $\mathcal{L}_{(\!|\cdot|\!)}$ term.

Most of the coercion typing rules have uniquely associated coercions. For example, ArrCoe is associated with the coercion `arr` and CovCoe is associated with the coercion `cov`. However, the base case coercion typing rules ReflDataCoe and RecVarCoe are each associated merely with the identity coercion `id` (at their respective types). These account for why the examples in Figure 2 require uses of `id` at the leaves of each coercion. While $\mathcal{L}_{(\!|\cdot|\!)}$ does not support universal quantification in types, these coercions are polymorphic primitives, where the type arguments are always inferred during type checking.

6 Type checking for $\mathcal{L}_{(\!|\cdot|\!)-}$

Let $\mathcal{L}_{(\!|\cdot|\!)-}$ be the language that coincides with $\mathcal{L}_{(\!|\cdot|\!)}$ except in eliding the explicit coercion construct, $c - a$, and introducing the following typing rule for implicit subtyping.

$$\textsc{Subtype}\ \frac{\Gamma \vdash A <:: A' \qquad \Gamma \vdash a : A}{\Gamma \vdash a : A'}$$

The comprehensive typing rules for $\mathcal{L}_{(\!|\cdot|\!)-}$ are compiled for reference in Appendix A. We develop $\mathcal{L}_{(\!|\cdot|\!)-}$ to be a version of $\mathcal{L}_{(\!|\cdot|\!)}$ where the burden of explicit coercions is alleviated by implicit inference of coercions.

In this section, we formalize a constraint-based type-checking algorithm for $\mathcal{L}_{(\!|\cdot|\!)-}$. Although type variables are introduced during constraint generation, their purpose is strictly confined to facilitating constraint resolution rather than

enabling general type inference. Because both λ-abstractions and *let*-bindings mandate explicit type annotations, all surface-level types are guaranteed to be ground. The development of a full type inference mechanism is left to future work.

The constraint generation algorithm, denoted $\mathsf{Generate}(\Gamma, a, A)$, generates a set of subtyping constraints necessary to check a term a against an expected type A under a typing context Γ. This algorithm follows a straightforward algorithmic application of the rules in Figure 7 by recursion on the structure of a. Each generated constraint corresponds to an implied application of the SUBTYPE rule required to construct a valid typing derivation; furthermore, each constraint is implicitly localized to the specific subterm at which subsumption is mandated. We omit a rigorous formalization of the coercion elaboration phase (the algorithmic insertion of coercions is fully realized in our implementation of type checking), however in order to reinforce that generated constraints correspond to specific sub-terms, we highlight the sub-term and the linked generated constraint with the same color in the rules of Figure 7.

Constraint generation for a term a in context Γ is performed by applying the constraint generation rules in Figure 7 recursively on the structure of a, which upon success produces a derivation of $\Gamma \vdash a : A \mid \mathcal{C}$. This is a usual typing judgment augmented with a constraint problem $\mathcal{C}$, where $\mathcal{C}$ is a set of constraints of the form $A <:: B$.

The constraint generation phase also performs type inference, however in contrast to usual type inference and checking procedures, this procedure cannot decide a term to be mal-typed. The constraint generation procedure will generate a constraint problem for any well-formed and well-scoped Γ, a, A. A derivation of $\Gamma \vdash a : A \mid \mathcal{C}$ justifies that a has type A *only if* the constraint problem $\mathcal{C}$ is solvable.

For example, constraint generation for `sub : Nat → Nat ⇒ Nat` (as defined *without* explicit coercions in figure 8) results in a superset of the following constraints, where all type variables are fresh. Each constraint is annotated with the particular sub-term of `sub` that it is linked to during constraint generation.

$$\{Nat\ X \underset{Zero}{<::} \mu Nat, \mu Nat \underset{sub'\ n'}{<::} Nat\ Y\}$$

As another example, constraint generation for `subs : NatList ⇒ μ NatList → NatList` (as defined *without* explicit coercions in figure 8) results in a superset of the following constraints, where all type variables are fresh. Each constraint is annotated with the particular sub-term of `subs` that it is linked to during constraint generation.

$$\{NatList\ X \underset{Nil}{<::} \mu NatList, \quad Nat \Rightarrow \mu Nat \underset{sub\ h1}{<::} \mu Nat \to \mu Nat\}$$

Lemma 4 (Termination of Generate). *$\mathsf{Generate}(\Gamma, a, A)$ terminates.*

Proof. Generate applies the rules of Figure 7 structurally recursively on the structure of a.

$$\textsc{Var}\ \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A \mid \varnothing}$$

$$\textsc{Con}\ \frac{X \text{ fresh} \qquad \langle D, \lambda Y. \cdots + k(\overline{A'_i\ \times}) + \ldots \rangle \in \mathscr{D} \qquad \overline{\Gamma \vdash\ a_i\ : A_i \mid \mathcal{C}_{a_i}}}{\Gamma \vdash k(\overline{a_i,}) : D\ Y \mid \bigcup_i (\mathcal{C}_{a_i} \cup \{\ A_i <:: A'_i[Y \mapsto X]\ \})}$$

$$\textsc{Lam}\ \frac{\Gamma, (x : A) \vdash b : B \mid \mathcal{C}_b}{\Gamma \vdash \lambda(x : A).b : A \to B \mid \mathcal{C}_b} \qquad \textsc{App}\ \frac{X \text{ fresh} \qquad \Gamma \vdash\ f\ : B \mid \mathcal{C}_f \qquad \Gamma \vdash a : A \mid \mathcal{C}_a}{\Gamma \vdash f\ a : X \mid \mathcal{C}_f \cup \mathcal{C}_a \cup \{\ B <:: A \to X\ \}}$$

$$\textsc{Gamma}\ \frac{\begin{array}{c} X, Y \text{ fresh} \qquad \langle D, \lambda X. \overline{k_i(\overline{A_{ij}\ \times})\ +} \rangle \in \mathscr{D} \\ \Gamma \vdash\ a\ : C \mid \mathcal{C}_a \qquad \overline{\Gamma, \overline{x_{ij} : A_{ij}[X \mapsto C]}, \vdash\ b_i\ : B_i \mid \mathcal{C}_{b_i}} \end{array}}{\Gamma \vdash \gamma a.\{\overline{k_i(\overline{x_{ij},}) \mapsto b_i\ }|\} : Y \mid \{\ C <:: D\ X\ \} \cup \mathcal{C}_a \cup \bigcup_i (\{\ B_i <:: Y\ \} \cup \mathcal{C}_{b_i})}$$

$$\textsc{Omega}\ \frac{\Gamma, (R_f : *), (f : R_f \to A), (x : D\ R_f) \vdash b : B \mid \mathcal{C}_b}{\Gamma \vdash \omega f(x).b : D \Rightarrow B \mid \mathcal{C}_b}$$

$$\textsc{Let}\ \frac{\Gamma \vdash\ a\ : A \mid \mathcal{C}_a \qquad \Gamma, (x : A') \vdash b : B \mid \mathcal{C}_b}{\Gamma \vdash \mathit{let}\ x : A' = a\ \mathit{in}\ b : B \mid \mathcal{C}_a \cup \{\ A <:: A'\ \} \cup \mathcal{C}_b}$$

Fig. 7: Constraint generation rules.

Lemma 5 (Soundness of Generate). *If σ is a solution to the constraint problem $\mathcal{C} = \mathit{Generate}(\Gamma, a, A)$, then $\Gamma \vdash_{\mathcal{L}_{(\!|\cdot|\!)}} a : A$*

Proof. Suppose that σ is a solution to the constraint problem $\mathcal{C} = \mathsf{Generate}(\Gamma, a, A)$ Then σ specifies a derivation for each constraint $A <:: B$ in $\mathcal{C}$. Use these derivations along with the rule Subtype to construct a derivation of $\Gamma \vdash_{\mathcal{L}_{(\!|\cdot|\!)-}} a : A$. Then, replace each use of Subtype in this typing derivation with the corresponding derivation of Coerce, as described in 5.4.

6.1 Constraint solving with Chronolog

We take a logic-programming approach to solving problems $\mathcal{C}$, where the coercions that justify the generated subtyping relations are calculated along the way. Our implementation uses a logic-programming engine, Chronolog, which adopts an alternative technique for constraint suspension to rule-level techniques in previous work [5, 12, 9]. Chronolog accepts a set of constraints, a logic variable

Algorithm 1 Constraint solving with Chronolog.

```
1: function Solve(C)
2:     σ ← ∅
3:     while C ≠ ∅ do
4:         match Chronolog(S, C) with
5:           case ⟨"failure"⟩ ⇒ return ⊥
6:           case ⟨"success", σ′⟩ ⇒ return σ′
7:           case ⟨"progress", σ′, C′⟩ ⇒
8:             σ ← Refine(C′) ∘ σ′
9:             C ← σ(C′)
10:        end match
11:    end while
12: end function
```

substitution, and a suspension meta-predicate S. Whenever a constraint that satisfies S appears, that constraint is immediately suspended. Suspended constraints are still refined by logic variable substitutions. Chronolog can terminate in three ways: (1) failure with unsolved constraints, (2) progress with suspended constraints, (3) success with no suspended constraints.

A particular choice of S is critical in this context due to the presence of the Roll and Unroll rules, which can be composed infinitely by straightforward SLD resolution, violating completeness of type checking. Chronolog prevents such infinite regresses by supporting a global suspension meta-predicate, S, where constraints that satisfy S are immediately suspended before being processed further. If such a refinement results in a constraint no longer satisfying S, then the constraint is resumed. We use a meta-predicate for S that holds of the following forms of constraints.

$$X <:: \mu D, \quad X <:: D\ A, \quad X <:: R, \quad \mu D <:: X, \quad D\ A <:: X, \quad R <:: X, \quad X <:: Y$$

Suspending constraint of these forms prevents Chronolog from applying rules in a cycle, which follows from a case analysis of all the possible rule applications to constraints not of suspended forms. In particular, the only forms of constraints that have a type variable as one side of the relation that are not suspended are those were the other side of the relation is either $D \Rightarrow A$ or $A \to B$. Consequently, Chronolog may attempt to resolve constraints of the form $X <:: A \to B$ utilizing either Arr or Cata, or attempt to resolve constraints of the form $A \Rightarrow B <:: X$ utilizing either Alg or Cata. This overlapping applicability inherently renders Chronolog a nondeterministic algorithm. Search branches that reach states where *no* rules may be applied are pruned with no backtracking. For type checking, we only need to check whether Solve results solutions or not. We implement Solve by repeatedly applying Chronolog and using Refine to refine any suspended constraints left over each iteration, as show in Algorithm 1.

Refine is defined by the following cases. When multiple cases of Refine are matched, any case can be chosen arbitrarily. Why this arbitrary choice preserves completeness is explained in the proof of Lemma 8.

$$\mathsf{Refine}(\mathcal{C}) := \begin{cases} \{X \mapsto R\} \text{ if } (X <:: R) \in \mathcal{C} \text{ or } (R <:: X) \in \mathcal{C} \\ \{X \mapsto \mu D\} \text{ if } (X <:: \mu D) \in \mathcal{C} \text{ or } (\mu D \in X) \in \mathcal{C} \\ \{X \mapsto \mu D\} \text{ if } (X <:: D\ A) \in \mathcal{C} \text{ or } (D\ A \in X) \in \mathcal{C} \\ \{X \mapsto Y\} \text{ if } (X <:: Y) \in \mathcal{C} \end{cases}$$

Lemma 6 (Decrease of Refine). *The number of unique variables in* $\mathsf{Refine}(\mathcal{C})(\mathcal{C})$ *is less than the number of unique variables in* $\mathcal{C}$*.*

Lemma 7 (Termination of Solve). $\mathsf{Solve}(\mathcal{C})$ *(Algorithm 1) terminates.*

Proof. Proof of termination proceeds by lexicographic induction on (s, v), where s is the size of $\mathcal{C}$ and v is the number of unique variables in $\mathcal{C}$. Each iteration of the **while**-loop in Solve begins with an invocation of $\mathsf{Chronolog}(S, \mathcal{C})$. If Chronolog results in $\langle$"failure"$\rangle$ or $\langle$"success"$\rangle$, then we are done. Otherwise, Chronolog results in $\langle$"progress", $\sigma', \mathcal{C}'\rangle$, in which case the new value assigned to $\mathcal{C}$ is $(\mathsf{Refine}(\mathcal{C}') \circ \sigma')(\mathcal{C}')$. Let (s', v') be the size and number of unique variables respectively in $(\mathsf{Refine}(\mathcal{C}') \circ \sigma')(\mathcal{C}')$. There are two cases:

1. If $\mathsf{Chronolog}(S, \mathcal{C})$ used *no* rule applications, then it must have merely suspended all goals, so $\mathcal{C}' = \mathcal{C}$, $\sigma = \mathrm{id}$, and $s' = s$. The constraints $(\mathsf{Refine}(\mathcal{C}') \circ \sigma')(\mathcal{C}') = (\mathsf{Refine}(\mathcal{C}'))(\mathcal{C}')$ has fewer unique variables than $\mathcal{C}$ by Lemma 6, so $v' < v$ and thus $(s', v') = (s, v') < (s, v)$.
2. If $\mathsf{Chronolog}(S, \mathcal{C})$ used *some* rule applications, then $s' < s$ by Lemma 1, and so $(s', v') < (s, v)$.

Lemma 8 (Completeness of Solve). *If there exists a solution to a constraint problem* $\mathcal{C}$*, then* $\mathsf{Solve}(\mathcal{C}) \neq \varnothing$*.*

Proof. Suppose there exists a solution to $\mathcal{C}$. Chronolog nondeterministically attempts every valid application of the algorithmic subtyping rules, which are complete by Theorem 3. Refine's handling of $X <:: R$ and $R <:: X$ constraints does not violate completeness since R is unrelated to any other type. Refine's handling of $X <:: \mu D$ and $\mu D <:: X$ constraints does not violate completeness since, any other appearances of X, once substituted for μD, may be rolled or unrolled by subsequent invocations of Chronolog to satisfy the constraints at those appearances. Refine's handling of $X <:: D\ A$ and $D\ A <:: X$ constraints does not violate completeness since, as there is an assumed solution to $\mathcal{C}$, A must be of the form $D\ (\ldots (D\ Y))$ (where Y is a distinct type variable from X) or $D\ (\ldots (\mu D))$, and so any other appearances of X may be rolled or unrolled by subsequent invocations of Chronolog to satisfy any other relations to the constraints at those appearances. The result follows since Solve terminates by Lemma 7.

Theorem 5 (Soundness of TypeCheck). *If* $\mathsf{TypeCheck}(\Gamma, a, A) = \top$ *(Algorithm 2) then* $\Gamma \vdash_{\mathcal{L}_{\langle\cdot\rangle-}} a : A$

Algorithm 2 Type checking for $\mathcal{L}_{(\!|\cdot|\!)-}$.

```
1: function TypeCheck(Γ, a, A)
2:     C ← Generate(Γ, a, A)
3:     return Solve(C) ≠ ∅
4: end function
```

Proof. Suppose $\mathsf{TypeCheck}(\Gamma, a, A) = \top$. Then there exists a solution to the set of constraints $\mathsf{Generate}(\Gamma, a, A)$, which are sufficient to demonstrate $\Gamma \vdash_{\mathcal{L}_{(\!|\cdot|\!)}} a : A$ (by Lemma 5).

Finally, formalize the relationship between $\mathcal{L}_{(\!|\cdot|\!)}$ and $\mathcal{L}_{(\!|\cdot|\!)-}$ in Theorem 6 in terms of erasure of explicit coercions $\mathcal{L}_{(\!|\cdot|\!)}$ to implicit coercions in $\mathcal{L}_{(\!|\cdot|\!)-}$ that are inferred during type checking.

Definition 1 (Erasure of terms from $\mathcal{L}_{(\!|\cdot|\!)}$ to $\mathcal{L}_{(\!|\cdot|\!)-}$). *Let a be a term of $\mathcal{L}_{(\!|\cdot|\!)}$. Then $|a|$ is the erasure of a into $\mathcal{L}_{(\!|\cdot|\!)-}$, which is defined by replacing in a each appearance of $c - a'$ with a'.*

Theorem 6 (Equivalence of $\mathcal{L}_{(\!|\cdot|\!)}$ and $\mathcal{L}_{(\!|\cdot|\!)-}$).

1. $\Gamma \vdash_{\mathcal{L}_{(\!|\cdot|\!)}} a : A \implies \Gamma \vdash_{\mathcal{L}_{(\!|\cdot|\!)-}} |a| : A$
2. $\Gamma \vdash_{\mathcal{L}_{(\!|\cdot|\!)-}} a : A \implies \exists a'.|a'| = a$ *and* $\Gamma \vdash_{\mathcal{L}_{(\!|\cdot|\!)}} a' : A$

7 Returning to the Examples

Recall the examples from Section 3 where `sub` and `subs` were implemented algebraically. Explicit coercions using `unroll`, `roll`, and `cata` were required to operate with the representation of datatypes as fixed points of signature functors and the functional interpretation of algebras via a catamorphism. Now, with the machinery of coercive subtyping in place, we can write those examples without any coercions, as demonstrated in Figure 8.

The type checking procedure described in previous sections collects and solves the implied subtyping constraints, which upon success determines a valid typing derivation for the term in $\mathcal{L}_{(\!|\cdot|\!)}$ with the inferred coercions explicitly inserted.

8 Related Work

Turner introduced the term *strong functional programming*, presented several arguments in favor of the approach, and proposed using structural termination as an *elementary* way to realize such a language [15]. Mendler proposed a recursor using an abstract type to ensure that recursive calls are permitted only on subdata for an inductive type [11]. The algebraic approach has been applied for modular implementation of interpreters [14]. Charity is a strong functional programming language based on categorical abstractions for initial algebras and final coalgebras [6]. Recursion is done via F-algebras, and there is no automatic

```
sub : μ Nat → Nat ⇒ μ Nat =
  λ (m : μ Nat) . ω sub'(n) . γ n .
    { Zero → m
    | Suc(n') → γ sub' n' .
      { Zero → Zero
      | Suc(m') → m' } }

subs : NatList ⇒ μ NatList → μ NatList =
  ω subs'(l1) . λ (l2 : μ NatList) . γ l1 .
    { Nil → Nil
    | Cons(h1, t1) → γ l2 .
      { Nil → Nil
      | Cons(h2, t2) →
          Cons(sub h1 h2, subs' t1 t2) } }
```

Fig. 8: Code for `sub` and `subs` with implicit coercions.

conversion from algebras to functions. This makes the code examples seem somewhat alien. Tofu uses the Calculus of Constructions as a language for defining terminating functions, again based on categorical ideas [17]. Charity and Tofu both support coinductive datatypes, which we did not study here.

Folding and unfolding signature functors is reminiscent of folding and unfolding recursive types as presented, for example, by Pierce [13]. The situation here is much simpler, though, because we consider the signature functors nominally. That is, recursive types μD and $\mu D'$ are never related in our theory, even if names D and D' are associated with the same underlying signature functor. In contrast, equirecursive types are treated extensionally, and this requires a greatest fixed-point algorithm to test equivalence. It is not our purpose to handle this more general situation. Rather, the goal is to handle standard inductive datatypes, which are usually treated nominally in functional programming. For example, two copies of the list datatype are not considered interchangeable in languages like Haskell or OCaml.

9 Conclusion and future work

Total algebraic programming can be made more feasible by inferring the basic coercions necessary for the method: rolling and unrolling signature functors, and turning an algebra into its catamorphism. Type inference generates subtyping constraints, whose solutions correspond to the needed coercions. A logic-programming engine processes constraints with respect to an algorithmic version of the subtyping rules. This algorithmic subtyping relation is proved equivalent to the declarative one, in Agda.

Furthermore, we mechanically verified (Lemma 4) that the algorithmic typing judgment admits unique derivations at ground types. While establishing the

mere satisfiability of the generated constraint problems (which include type variables) is sufficient for the purposes of type checking, we defer to future work the demonstration of a suitable equivalence relation that characterizes the solutions produced by Solve.

Future work also includes richer forms of algebraic programming. One powerful generalization of structural recursion is to recursive coalgebras [4]. This encompasses divide-and-conquer algorithms, which are beyond the scope of structural recursion. A different approach to divide-and-conquer recursion has also been considered by [1]. There, algebras call functions to pass between various abstract types. These would be excellent candidates for inference using coercive subtyping, as we proposed in this paper. Interestingly, there are additional coercions from the types $D \Rightarrow X$ of algebras that would arise from applying our approach to that work. This provides further motivation for maintaining algebras as separate syntactic constructs, and inferring uses of *cata*. It would also be interesting to infer coercions for coalgebraic programming with coinductive types.

Another avenue of future work is to prove *coherence* for $\mathcal{L}_{(\!|\cdot|\!)}$. This property states that if $\Gamma \vdash_{\mathcal{L}_{(\!|\cdot|\!)}} a_1 : A$ and $\Gamma \vdash_{\mathcal{L}_{(\!|\cdot|\!)}} a_2 : A$, and if $|a_1| = |a_2|$, then a_1 and a_2 are semantically equivalent. That is, inserting coercions in different places in the same term does not change it semantics. We conjecture this is true, but proving this property can be technically involved (cf. [3]).

Acknowledgments. Thanks to the anonymous reviews for very helpful detailed feedback, which has greatly improved the paper.

A Typing rules for $\mathcal{L}_{(\!|\cdot|\!)\_}$

$$\textsc{Var}\ \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Con}\ \frac{\langle D, \lambda X. \cdots + k(\overline{A_i\ \times}) + \cdots \rangle \in \mathscr{D} \qquad \overline{\Gamma \vdash a_i : A_i[X \mapsto C]}}{\Gamma \vdash k(\overline{a_i,}) : D\ C}$$

$$\textsc{Lam}\ \frac{\Gamma, (x : A) \vdash b : B}{\Gamma \vdash \lambda(x : A).b : A \to B} \qquad \textsc{App}\ \frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B}$$

$$\textsc{Omega}\ \frac{\Gamma, (R_f : *), (f : R_f \to A), (x : D\ R_f) \vdash b : B}{\Gamma \vdash \omega f(x).b : D \Rightarrow B}$$

$$\textsc{Gamma}\ \frac{\langle D, \lambda X. k_i(\overline{A_{ij}\ \times}) + \rangle \in \mathscr{D} \qquad \Gamma \vdash a : D\ C \qquad \overline{\Gamma, \overline{x_{ij} : A_{ij}[X \mapsto C]}, \vdash b_i : B}}{\Gamma \vdash \gamma a.\{\overline{k_i(\overline{x_{ij},}) \mapsto b_i\ |}\} : B}$$

$$\textsc{Let}\ \frac{\Gamma, (x : A) \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathit{let}\ x : A = a\ \mathit{in}\ b : B} \qquad \textsc{Subtype}\ \frac{\Gamma \vdash A <:: A' \qquad \Gamma \vdash a : A}{\Gamma \vdash a : A'}$$

References

1. Pedro Abreu, Benjamin Delaware, Alex Hubers, Christa Jenkins, J. Garrett Morris, and Aaron Stump. A type-based approach to divide-and-conquer recursion in coq. *Proc. ACM Program. Lang.*, 7(POPL):61–90, 2023.
2. Krzysztof R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *J. ACM*, 29(3):841–862, July 1982.
3. Dariusz Biernacki and Piotr Polesiuk. Logical relations for coherence of effect subtyping. *Log. Methods Comput. Sci.*, 14(1), 2018.
4. Venanzio Capretta, Tarmo Uustalu, and Varmo Vene. Recursive coalgebras from comonads. *Inf. Comput.*, 204(4):437–468, 2006.
5. K. L. Clark, F. G. McCabe, and S. Gregory. Ic-prolog language features. In K. L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 254–266. Academic Press, London, 1982.
6. Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Department of Computer Science, The University of Calgary, June 1992.
7. J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *J. ACM*, 24(1):68–95, January 1977.
8. Tatsuya Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, 1987.
9. Michael Hanus. Improving residuation in declarative programs. In *Practical Aspects of Declarative Languages: 21st International Symposium, PADL 2019, Lisbon, Portugal, January 14-15, 2019, Proceedings*, pages 82–97, Berlin, Heidelberg, 2019. Springer-Verlag.
10. Robert Kowalski. Predicate logic as programming language. volume 74, pages 569–574, 01 1974.
11. N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1):159 – 172, 1991.
12. Lee Naish. An introduction to MU-PROLOG. Technical Report 82/2, University of Melbourne, Department of Computer Science, 1982.
13. Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.
14. Wouter Swierstra. Data Types à La Carte. *J. Funct. Program.*, 18(4):423–436, July 2008.
15. D. A. Turner. Elementary Strong Functional Programming. In *Proceedings of the First International Symposium on Functional Programming Languages in Education*, FPLE '95, page 1–13, Berlin, Heidelberg, 1995. Springer-Verlag.
16. Tarmo Uustalu and Varmo Vene. Mendler-style Inductive Types, Categorically. *Nordic J. of Computing*, 6(3):343–361, September 1999.
17. Baltasar Trancón y Widemann. Tofu - towards practical church-style total functional programming. In *Workshop der GI-Fachgruppe Programmiersprachen und Rechenkonzepte, Bad Honnef, 3.-5. Mai*, 2010. Available at https://www-ps.informatik.uni-kiel.de/fg214/Honnef2010/TechnischerBericht1010.pdf.