Component-Oriented Synthesis via Algebras and Combinatory Logic

JAN BESSAI and JAKOB REHOF, Technical University Dortmund
BORIS DÜDDER, University of Copenhagen
GEORGE T. HEINEMAN, Worcester Polytechnic Institute

Algebraic techniques are used to generalize component-oriented synthesis to a target language agnostic setting. Components are operations of algebraic signatures supporting subtype and parametric polymorphism augmented with semantic specifications. Component implementations are formalized using algebras and thus are programming language neutral. For synthesis, the paper presents formally verified type checking and enumerative type inhabitation algorithms for combinatory logic with intersection types, finite well-formed substitution spaces, and distributing covariant constructors. These algorithms provide sound, complete, and unique enumerations of all objects generated by any given target language algebra using translations from, and to, combinatory logic. Expressiveness is studied using the example of Mini-ML-Box, a staged type system corresponding to intuitionistic modal logic S4. The resulting framework is implemented in Coq and Scala. It is evaluated in experiments that automatically create product line members from Scala combinators encapsulating pre-existing Java code.

CCS Concepts: • **Computing methodologies → Combinatorial algorithms**; • **Software and its engineering → Automatic programming**; • **Theory of computation** → *Algebraic language theory*;

General Terms: Design, Languages, Theory

Additional Key Words and Phrases: Program Synthesis, Type Theory, Combinatory Logic, Grammar, Programming Languages

ACM Reference format:
Jan Bessai, Jakob Rehof, Boris Düdder, and George T. Heineman. 2017. Component-Oriented Synthesis via Algebras and Combinatory Logic. *Proc. ACM Program. Lang.* 99999, 99999, Article 99999 (2017), 27 pages.
DOI: 0000001.0000001

1 INTRODUCTION

The core idea of this paper is simple: Specify an interface, implement it, and then ask an algorithm to produce all possible programs of a given type using just the components of that interface. This idea can be realized independently of the programming language involved. To this end, interfaces are generalized using the well-known formalism of algebraic specification with signatures (Böhm and Berarducci 1985; Burghardt 2002; Ehrig and Mahr 1985; Gogolla 1984; Hoare, C. A. R. 1972; Jacobs and Rutten 1997; Padawitz 2011; Srinivas, Yellamraju V. and Jüllig, Richard 1995; Wirsing 1986). Programs are produced using type inhabitation for combinatory logic with intersection types (Düdder et al. 2014, 2012; Rehof and Urzyczyn 2011) and an algebraic translation. Any interface specified by an algebraic signature can be translated into a combinatory logic context Γ. This automatic translation does not need any additional user input. The type system – combinatory logic with intersection types, distributing covariant constructors, and finite substitution spaces – is powerful enough to encode signature operations with parametric and subtype polymorphism. A type inhabitation algorithm is presented which enumerates all terms typable by a given type under the assumptions in Γ. Intersection types are used to specify an additional layer of semantics, which is independent of the signature specification. This ephemeral layer adds documentation and

DOI: 0000001.0000001

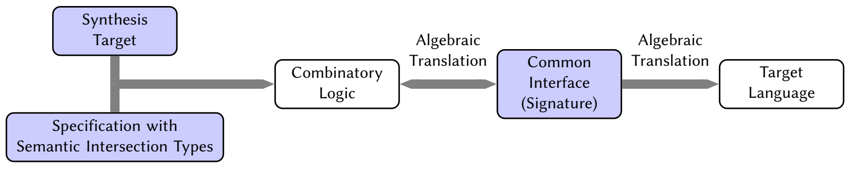

Fig. 1. Framework Overview

does not require a formal mapping to implementations. Representing a Turing-complete programming language (Rehof 2013), it can control composition and is most useful to eliminate unwanted solutions from the enumerated results. Terms in combinatory logic are proven to be translatable to any given signature implementation in any target programming language. An overview of the resulting framework is depicted in Fig. 1, where shaded boxes indicate parts requiring user input. Application to Mini-ML$^{\Box}$ as target language is studied to demonstrate the expressive power. The framework has been implemented and proven in 17k lines of Coq. Practical experiments have been carried out using an additional implementation in Scala (5k lines). Sources are publicly available[1].

1.1 Example

To get an insight into the functionality of the framework, let us start with a journey to a strange modern kingdom, implemented in Scala (Listing 1), where our egg friend Humpty Dumpty suffered an unfortunate side effect when falling from a wall. Right in front of us, we see a puddle of liquid (line 2). For now, we can ignore the exact contents of the puddle and specify that picking it up (with method `apply`) will give us a liquid. We know this puddle is the whipped-egg contents of poor Humpty Dumpty. Cataloging our kingdom, we annotate such observations as additional type information in fields named `semanticType`. Next to the puddle we meet a local ornithologist (line 3), an expert on avian living, who tells us that some thieving birds stole the head pieces of Humpty Dumpty's broken shell and flew to the east, while others took his bottom shell pieces to the west. We also learn where the queen's crown jewels and the princess went. We combine this information using the intersection type operator `:&:`, expressing that a single value is of multiple types. On our way to the castle, we meet an armed revenue officer (line 8) who will (if necessary by force) collect precious objects from any living thing. We observe him to be a prudent servant of the state, requiring proof of the tax status, which depends on the kingdom direction and possession, before he returns exactly the taxable possession. We will write down function types at the semantic level using a different arrow symbol `=>:` and curry them when there are multiple arguments. Unsure of what exactly the taxable possessions are, we add them as a semantic type variable (line 40). Walking through the district of craftsmen, we meet a modern-day alchemist (line 12) testing his new centrifuge. It can turn any liquid into a pair of liquids, as his tests with egg white and egg yolk have proven. His first test subject was a gift from the neighboring school of the arts of fine pastry. This pastry school offers two analytical and one applied program of study. Analytical scientists (line 16) can tell two things apart, returning the one thing they are specialized on. The other thing can be of any type because it will be ignored; we indicate this by `Omega`. The analytical programs at the pastry school focus on meringue (line 20) and biscuit (line 23), one of which requires experts on egg white, while the other requires experts on egg yolk. The applied program educates masters in filling any sort of pastry with liquids (line 26). Usually they fill doughnuts, but given their skill, they could surely fill a glued together Humpty Dumpty egg shell with egg white and then with egg yolk - even if the intermediate zombie certainly is gruesome to look at. Approaching the castle, we see the repair shop of the royal hot glue advisor who can fix all sorts of collections of precious objects by gluing them together. The description of these things may vary, so we again add it as a type variable (line 41). At the closed castle doors, we find the new tax code. Compensating for the expensive loss of crown jewels, there is a new tax on air dwellers in every direction. We write this down

[1]See supplemental material. Link to be included in the final version.

using a taxonomy (subtype rules) for the `Taxable` and `Direction` types (line 36), which previously described what the armed revenue office (line 8) was able to collect. Instead of the planned castle visit, we will now use what we have learned so far, writing down a piece of code (Listing 2) that will reincarnate our broken friend back to life. We first get an object to address the kingdom described above (line 2). Then we restrict the type variables to range over not just any types, but only those which might be helpful. For the armed revenue officer, it would be great if he could get back the stolen shell pieces via tax collection, hence we add their types as options for the instantiation of the variable `possession` (line 4). This restriction is called kinding, because it restricts the type of types. The kinding is merged with a restriction for descriptions of objects that the hot glue advisor should be able to glue together. If he has any part in restoring Humpty Dumpty, he should be able to reconnect the broken head and body shell (line 5). We then start the framework machinery, on a reflected version of the kingdom (line 6), that additionally gets the taxonomy and kinding information. Reflection searches for operations annotated by `@combinator` to construct a signature algebra for them. For now, we can think of signature algebras as interface descriptions; their exact mathematical definition will follow soon. To reincarnate Humpty Dumpty we call inhabit on the reflected kingdom (line 7), asking for an instance of Scala type `Any`, which is semantically described by the type `HumptyDumpty`. Of all the (potentially many) possible ways of reincarnation, we choose the first one and return it as a call tree (line 8). Method `reincarnate` then returns our friend back from the dead (line 9).

Calling `reincarnate` would use all the helper implementations we avoided to provide (indicated by “???”), so we can just inspect the reincarnation call tree to convince ourselves of success (Listing 3). The royal glue advisor is asked to repair the egg shells, which the armed revenue officer collects at the positions specified by the ornithologist. This will give us a glued together precious object that is then passed to the filled pastry master who will fill it with egg white. Egg white is obtained by centrifuging the puddle and using the analytic skills of a meringue master. The first filling process returns the zombie so we pass it to the filled pastry master again to reinsert the egg yolk. The latter is obtained by centrifuging the puddle and using the biscuit master. From the filled pastry master, we get back an instance of Scala type `Any`, which is semantically described by `HumptyDumpty`. The reader is invited to take a journey investigating in depth what now seems like magic:

Section 2 Within the dry desert landscapes of abstract math we will find the definition of sigma algebras and examine intersection types. Then the two are connected, resulting in a formal target language agnostic synthesis framework. We establish guarantees for soundness, completeness, and uniqueness of results.

Section 3 Demonstrating application to a well-understood formal system, we learn how to synthesize terms typed in the simply typed modal ML language Mini-ML$^{\Box}$.

Section 4 We return to the more hospitable landscape of Scala programs, which as we have seen, can also be synthesized. Some details of a practical application are explained. The implementation is further evaluated using synthesis of a product line of solitaire games from Java ASTs wrapped in Scala combinators.

Sections 5, 6 We end our journey learning about interesting places to visit next and reflecting on our travel.

2 THEORY

We start with basic definitions for algebraic signatures (Sect. 2.1), type theoretic results for combinatory logic with intersection types, distributing covariant constructors, and finite well-formed substitution spaces (Sect. 2.2), and then explain the connection between the algebraic and type theoretic world (Sect. 2.3). Notations throughout this paper are consistent with the category **Set**. Here products $\prod_{i \in I} X_i$ coincide with cartesian products and sums $\sum_{i \in I} X_i$ are disjoint unions of pairs (i, X_i) for $i \in I$. For product projections, we write π_i, or when reading the first and second component of a sum, π and π' respectively. Formal proofs have been carried out in Coq, where products are dependent functions `forall s, P s` and sums are sigma or existential types (depending on the summand being in `Prop` or `Type`). No axioms additional to those built into Coq 8.5 have been used, therefore results should generalize to locally cartesian closed categories (Seely 1984), categories with families (Hofmann 1997) and other structures capable of models for dependent type theory (Curien et al. 2014).

```
class Kingdom {
  @combinator object Puddle { def apply(): Liquid = ???; val semanticType = 'WhippedEgg  }
  @combinator object Ornithologist {
    def apply(): Living = ???
    val semanticType = 'AirDweller('South, 'CrownJewels) :&: 'AirDweller('North, 'Princess) :&:
      'AirDweller('East, 'Pieces('HeadShell)) :&: 'AirDweller('West, 'Pieces('BottomShell))
  }
  @combinator object ArmedRevenueOfficer {
    def apply(targetGroup: Living): Seq[PreciousObject] = ???
    val semanticType = 'Taxable('Direction, possession) =>: possession
  }
  @combinator object CentrifugeAlchemist {
    def apply(l: Liquid): (Liquid, Liquid) = ???
    val semanticType = 'WhippedEgg =>: 'Pair('EggWhite, 'EggYolk)
  }
  abstract class Analyst[A](specialty: Type) {
    def apply(objects: (A, A)): A
    val semanticType = ('Pair(Omega, specialty) =>: specialty) :&: ('Pair(specialty, Omega) =>: specialty)
  }
  @combinator object MeringuePastryMaster extends Analyst[Liquid]('EggWhite) {
    def apply(eggFilling: (Liquid, Liquid)): Liquid = ???
  }
  @combinator object BiscuitPastryMaster extends Analyst[Liquid]('EggYolk) {
    def apply(eggFilling: (Liquid, Liquid)): Liquid = ???
  }
  @combinator object FilledPastryMaster {
    def apply(pastry: Any, filling: Liquid): Any = ???
    val semanticType = ('Bagel =>: 'Marmelade =>: 'Doughnut) :&:
      ('HeadShell :&: 'BottomShell :&: 'Glued =>: 'EggWhite =>: 'HumptyZombie) :&:
      ('HumptyZombie =>: 'EggYolk =>: 'HumptyDumpty)
  }
  @combinator object RoyalHotGlueAdvisor {
    def apply(parts: Seq[PreciousObject]): PreciousObject = ???
    val semanticType = 'Pieces(description) =>: description :&: 'Glued
  }
  lazy val taxonomy =
    Taxonomy("Direction").addSubtype("North").addSubtype("East").addSubtype("South").addSubtype("West")
      .merge(Taxonomy("Taxable").addSubtype("AirDweller"))

  lazy val possession = Variable("possession")
  lazy val description = Variable("description") }
```

Listing 1. Scala code describing a fairy tale kingdom

```
object HumptyDumpty {
  lazy val kingdom = new Kingdom
  lazy val kinding =
    Kinding(kingdom.possession).addOption('Pieces('HeadShell)).addOption('Pieces('BottomShell))
      .merge(Kinding(kingdom.description).addOption('HeadShell).addOption('BottomShell))
  lazy val reflectedKingdom = ReflectedRepository(kingdom, kingdom.taxonomy, kinding)
  lazy val allReincarnations = reflectedKingdom.inhabit[Any]('HumptyDumpty)
  lazy val reincarnationPlan = allReincarnations.terms.index(0)
  def reincarnate() = allReincarnations.interpretedTerms.index(0) }
```

Listing 2. Scala code to reincarnate Humpty Dumpty by type inhabitation

```
Tree(FilledPastryMaster,
  List(Tree(FilledPastryMaster,
    List(Tree(RoyalHotGlueAdvisor,List(Tree(ArmedRevenueOfficer,List(Tree(Ornithologist,List())))))),
         Tree(MeringuePastryMaster,List(Tree(CentrifugeAlchemist,List(Tree(Puddle,List()))))))),
    Tree(BiscuitPastryMaster,List(Tree(CentrifugeAlchemist,List(Tree(Puddle,List())))))))
```

Listing 3. Reincarnation call tree for Humpty Dumpty

2.1 Algebraic Signatures

Signatures are a common language for interfaces and connect combinatory logic with synthesis target languages.

Definition 2.1 (Term Signatures). Given a map $\mathbb{S} : \mathsf{Set} \to \mathsf{Set}$ and a finite set of variables $\mathbb{V}$, the term signature $\Sigma_{\mathbb{S},\mathbb{V}} = (\mathbb{O}, \mathsf{arity}, \mathsf{domain}, \mathsf{range})$ is defined by:

- A set $\mathbb{O}$ of operation symbols
- A function $\mathsf{arity} : \mathbb{O} \to \mathbb{N}^0$
- A family $\mathsf{domain} : (\prod_{i=1}^{\mathsf{arity}(op)} \mathbb{S}(\mathbb{V}))_{op \in \mathbb{O}}$
- A function $\mathsf{range} : \mathbb{O} \to \mathbb{S}(\mathbb{V})$

We call $s \in \mathbb{S}(\emptyset)$ a closed sort and require closed sorts to be countable. Open sorts $s \in \mathbb{S}(\mathbb{V})$ are used to provide parametric polymorphism and closed sorts can be created from open sorts by applying substitutions.

Definition 2.2 (Substitution). Given a map $\mathbb{S} : \mathsf{Set} \to \mathsf{Set}$ and a set of variables $\mathbb{V}$, we lift substitutions $S : \mathbb{V} \to \mathbb{S}(\emptyset)$ to $S^+ : \mathbb{S}(\mathbb{V}) \to \mathbb{S}(\emptyset)$ by applying a substitution map $\_^+ : (\mathbb{V} \to \mathbb{S}(\emptyset)) \to \mathbb{S}(\mathbb{V}) \to \mathbb{S}(\emptyset)$.

In the following, we implicitly assume that a substitution map exists for all sort structures $\mathbb{S}$ under consideration. As a special case this is true for all monads $(\mathbb{S}, \eta, \mu)$, where $\_^+$ can be implemented by the monadic bind operator ($\texttt{>>=}\ f = \mu \circ \mathbb{S}(f)$), which is well known from Haskell. Closed sorts can be (pre-)ordered to model subtype polymorphism of implementation languages.

Definition 2.3 (Sort Preorders). Given a map $\mathbb{S} : \mathsf{Set} \to \mathsf{Set}$, a sort preorder is a relation $\leq_{\mathbb{S}} \subseteq \mathbb{S}(\emptyset) \times \mathbb{S}(\emptyset)$ that is reflexive (for all $s \in \mathbb{S}(\emptyset)$, $s \leq_{\mathbb{S}} s$) and transitive (for all $s_1, s_2, s_3 \in \mathbb{S}(\emptyset)$, if $s_1 \leq_{\mathbb{S}} s_2$ and $s_2 \leq_{\mathbb{S}} s_3$ then $s_1 \leq_{\mathbb{S}} s_3$).

Subsorted $\Sigma_{\mathbb{S},\mathbb{V}}$-Algebras connect interfaces (signatures) with implementations (carrier elements).

Definition 2.4 (Subsorted $\Sigma_{\mathbb{S},\mathbb{V}}$-Algebra). Given a term signature $\Sigma_{\mathbb{S},\mathbb{V}} = (\mathbb{O}, \mathsf{arity}, \mathsf{domain}, \mathsf{range})$, a sort preorder $\leq_{\mathbb{S}}$ and a function space $\mathsf{WF} \subseteq \mathbb{V} \to \mathbb{S}(\emptyset)$ of well-formed substitutions, a subsorted $\Sigma_{\mathbb{S},\mathbb{V}}$-Algebra consists of

- A carrier $\mathbb{C}$, which is a family of sets indexed by closed sorts $s \in \mathbb{S}(\emptyset)$
- A $\mathbb{S}(\emptyset)$ indexed family of morphisms $m_s : \mathbb{F}_s(\mathbb{C}) \to \mathbb{C}_s$ where

$$\mathbb{F}_s(\mathbb{C}) = \sum_{S \in \mathsf{WF}} \sum_{o \in \mathsf{ranged}(s,S)} \prod_{i=1}^{\mathsf{arity}(o)} \mathbb{C}_{S^+(\pi_i(\mathsf{domain}_o))}$$

with $\mathsf{ranged} : \mathbb{S}(\emptyset) \times \mathsf{WF} \to \mathsf{Set}$ defined by $\mathsf{ranged}(s, S) = \{o \in \mathbb{O} \mid S^+(\mathsf{range}(o)) \leq_{\mathbb{S}} s\}$.

Intuitively, $\mathbb{C}_s$ can represent a program in the target language, which encodes an entity of sort s. When applying the algebra morphism family m, we choose a sort s, a substitution S and an operation o from the signature, such that executing o results in s. Next, we have to supply arguments for each of the parameters of o. The sort of the i^{th} argument is S lifted to close the open i^{th} sort selected from domain_o via projection π_i. Mathematically, the choices are encoded in the sums and products of $\mathbb{F}_s(\mathbb{C})$, so we can write them down as the triple (S, o, args). The result of the algebra is an object $\mathbb{C}_s$ representing sort s. Categorically, $\mathbb{F}$ induces an endofunctor $\mathsf{Set}^{\mathbb{S}(\emptyset)} \to \mathsf{Set}^{\mathbb{S}(\emptyset)}$ where $\mathsf{Set}^{\mathbb{S}(\emptyset)}$ is the category of $\mathbb{S}(\emptyset)$-indexed sets (Padawitz 2011). In $\mathsf{Set}^{\mathbb{S}(\emptyset)}$, objects are sort indexed sets like $\mathbb{C}$, and morphisms from $\mathbb{C}$ to $\mathbb{D}$ are indexed families of functions $f : (\mathbb{C}_s \to \mathbb{D}_s)_s$. The induced functor $\mathbb{F}$ can then be defined by $\mathbb{F}(\mathbb{C})_s = \mathbb{F}_s(\mathbb{C})$ and $\mathbb{F}(f)_s((S, o, \mathsf{args})) = (S, o, \mathsf{args}')$ where $\pi_i(\mathsf{args}') = f_{\pi_i(\mathsf{domain}_o)}(\pi_i(\mathsf{args}))$ for all $i \in \{1, \ldots, \mathsf{arity}(o)\}$. Note that leaving out variables and substitutions ($\mathbb{V} = \emptyset$), we can obtain subsorted Σ-Algebras as described by Gogolla (1984). Further, by choosing $\leq_{\mathbb{S}} = \{(s, s) \mid s \in \mathbb{S}(\emptyset)\}$, we can obtain the usual notion of a many sorted Σ-Algebra (Padawitz 2011). Restricting $\mathbb{S}$ to a singleton set $\{s\}$ yields $\mathbb{F}_s(\mathbb{C}) = \sum_{o \in \mathbb{O}} \prod_{i=1}^{\mathsf{arity}(o)} \mathbb{C}_s \to \mathbb{C}_s$, the standard formulation of single sorted Σ-Algebras (Ehrig and Mahr 1985; Jacobs and Rutten 1997).

Example 2.5. The following algebraic description can be used for even and odd natural numbers:

$$\begin{aligned}
&\mathbb{V} = \{\alpha, \beta\} \quad \mathbb{S}(X) = X \uplus \{\text{Even}, \text{Odd}, \text{Nat}\} \quad \mathbb{O} = \{\mathsf{one}, \mathsf{succ}\}\\
&\mathsf{arity} = \{\mathsf{one} \mapsto 0, \mathsf{succ} \mapsto 1\} \quad \mathsf{domain} = \{\mathsf{one} \mapsto (), \mathsf{succ} \mapsto \alpha\} \quad \mathsf{range} = \{\mathsf{one} \mapsto \text{Odd}, \mathsf{succ} \mapsto \beta\}\\
&\text{WF} = \{\{\alpha \mapsto \text{Odd}, \beta \mapsto \text{Even}\}, \{\alpha \mapsto \text{Even}, \beta \mapsto \text{Odd}\}\}\\
&\mathbb{C}_{\text{Odd}} \ni o ::= \mathtt{I} \mid \mathtt{I}e;\ \mathbb{C}_{\text{Even}} \ni e ::= \mathtt{I}o;\ \mathbb{C}_{\text{Nat}} \ni n ::= e \mid o\\
&m_{\text{Even}} = \{(\{\alpha \mapsto \text{Odd}, \beta \mapsto \text{Even}\}, \mathsf{succ}, o) \mapsto \mathtt{I}o \mid o \in \mathbb{C}_{\text{Odd}}\}\\
&m_{\text{Odd}} = \{(S, \mathsf{one}, ()) \mapsto \mathtt{I} \mid S \in \text{WF}\} \cup \{(\{\alpha \mapsto \text{Even}, \beta \mapsto \text{Odd}\}, \mathsf{succ}, e) \mapsto \mathtt{I}e \mid e \in \mathbb{C}_{\text{Even}}\}\\
&m_{\text{Nat}} = m_{\text{Even}} \cup m_{\text{Odd}}
\end{aligned}$$

Here, the well-formedness of substitutions WF is crucial to ensure that domain and range parities of *succ* are different. In a more concise notation we may write down the signature as: $\Sigma_{\mathbb{S},\mathbb{V}} = \{\mathsf{one} : () \twoheadrightarrow \text{Odd}; \mathsf{succ} : \alpha \twoheadrightarrow \beta\}$ with $\twoheadrightarrow$ acting as separator between domain and range. In this example, lifting substitution S just means a domain extension, hence we get $S^+ = \{v \mapsto S(v) \mid v \in \mathbb{V}\} \cup \{s \mapsto s \mid s \in \{\text{Even}, \text{Odd}, \text{Nat}\}\}$. Subsorting expresses that every even or odd number is also natural, $\leq_{\mathbb{S}} = \{(\text{Even}, \text{Even}), (\text{Odd}, \text{Odd}), (\text{Nat}, \text{Nat}), (\text{Even}, \text{Nat}), (\text{Odd}, \text{Nat})\}$. A possible target language for interpretation of the signature are strings signifying unary encoded numbers. These can be described by the context free grammar given above. It includes terminal $\mathtt{I}$ and nonterminals o, e and n for odd, even, and natural numbers. Sets of words with parse trees rooted in those symbols then act as the carriers.

For the algebra, morphism m_{Even} is restricted to inputs succ and substitution $S = \{\alpha \mapsto \text{Odd}, \beta \mapsto \text{Even}\}$, because there is no other combination to obtain $S^+(\mathsf{range}(o)) \leq_{\mathbb{S}} \text{Even}$. Additionally to succ and S with toggled parities, m_{Odd} includes cases for operation one and any well-formed substitution. Morphism m_{Nat} combines all possible choices. We are free to specify results for the mappings. While any element of $\mathbb{C}_{\text{Even}}$ would be a valid result of m_{Even}, we choose the obviously meaningful string "$\mathtt{I}o$" for whatever odd number o we get as input. This freedom of choice is important, because later algebras will be part of the framework input. There are two ways to limit possibilities to specify nonsensical algebras. Either additional (in-)equations can be stated and checked independently of the framework or more sorts can be added, allowing for higher granularity in carrier sets. Here, we use the second possibility when compared to the modeling over a single sort Nat, presented in Ex. 5.4 of Jacobs and Rutten (1997): parities indicated by sorts are guaranteed to match the number of "$\mathtt{I}$"-Symbols, whenever morphisms map into their specified carrier sets.

2.2 Type System

We proceed by defining a type system for synthesis via the type inhabitation problem in combinatory logic. In combinatory logic, well-typed terms can only be formed by applying combinators (signature operations) from a type context Γ. This is in contrast to lambda calculus, which allows to introduce new lambda abstractions with an abstraction rule ($\rightarrow$ I). We can however simulate an abstraction rule by adding appropriately typed S, K and I combinators to the type context, which results in an SKI-calculus equivalent to the λ-calculus (Dezani-Ciancaglini and Hindley 1992). For our purposes, contexts only include operations from signatures, which most often will not be S, K and I. All type system rules and notational conventions are stated in Fig. 2. They are an extended form of the type system presented in (Düdder et al. 2012). To later avoid unnecessary notational clutter, we identify the symbols for sets of combinators with the symbols for signature operations. Type variables are drawn from a set $\mathbb{V}$ and constructor symbols from $\mathbb{C}$, where each constructor symbol C has a fixed arity given by $|C|$. Type schemes $\mathbb{T}_{\mathbb{V}}$ are formed over variables, applied constructors, the special constant ω and arrows or intersections between type schemes. Types are closed type schemes without variables. Contexts Γ assign type schemes to operations. Using S^* we can lift substitutions $S : \mathbb{V} \rightarrow \mathbf{T}$, which are maps from variables to closed types, to maps from type schemes to closed types. As with algebras, we choose an arbitrary fixed restriction WF on the function space, from which substitutions S may be drawn.

Constructor symbols of the same arity are preordered by $\leq_{\mathbb{C}}$, which is an arbitrary reflexive transitive binary relation. In a later stage of modeling, $\leq_{\mathbb{C}}$ will be chosen in accordance with $\leq_{\mathbb{S}}$. Preorders $\leq_{\mathbb{C}}$ are extended to preorders $\leq \subset \mathbf{T} \times \mathbf{T}$ on types. This extension is standard for intersection types (Barendregt et al. 1983) with the notable exception of rule (CDist), which allows to distribute intersections over constructor symbols. Subtyping is idempotent ($\mathsf{a} \leq \mathsf{a} \cap \mathsf{a}$) and we have ω as universal type with $\mathsf{a} \leq \omega \leq \omega \to \omega \leq \omega$.

Type system rules include arrow elimination ($\to$ E) to form terms by application and subtyping ($\leq$) according to the type preorder $\leq$. Rule ($\cap$I) allows to deduce that term M has type $\mathsf{a} \cap \mathsf{b}$, denoted by $\Gamma \vdash M : \mathsf{a} \cap \mathsf{b}$, if there are proofs that M has both type a and type b. Rule (Var) allows to assign type a to a combinator x, if there is a well formed substitution S on $\Gamma(x)$ resulting in the closed type a. Bounded combinatory logic with bound k (BCL_k) known from (Düdder et al. 2012) is a restriction of the type system presented here. It is obtained by fixing all constructors to be of arity 0 and choosing WF to make substitutions adhere to a level-restriction allowing a maximal depth of k nested arrows in substitution results. Standard intersection type systems (Dezani-Ciancaglini and Hindley 1992) model contexts as relations and explicitly include the rule $\dfrac{}{\Gamma \vdash M : \omega}\ (\omega)$.
With contexts being functions, rule (ω) can be shown to be admissible for all non-empty spaces of well-formed substitutions. For proofs and later definitions, it is important to define the concept of paths.

Definition 2.6 (Path). Paths $p, p' \in \mathbb{P}$ are types defined by the grammar:

$$\mathbf{T} \supset \mathbb{P}_{\mathbb{C}} \ni p_c ::= C_1() \mid C_2(\omega, \dots, \omega, p, \omega, \dots, \omega) \qquad \mathbf{T} \supset \mathbb{P} \ni p, p_1, \dots, p_n ::= p_c \mid \mathsf{a} \to p$$

where $\omega, \dots, \omega$ denotes arbitrary, possibly empty, finite sequences of ω and arities of constructors C_1 and C_2 respected. The length of a path is computed by $||\cdot|| : \mathbb{P} \to \mathbb{N}^0$, defined by

$$||p|| = \{p \mapsto 0 \mid p \in \mathbb{P}_{\mathbb{C}}\} \uplus \{\mathsf{a} \to p_1 \mapsto 1 + ||p_1|| \mid \mathsf{a} \in \mathbf{T} \text{ and } p_1 \in \mathbb{P}\}$$

A path p can be deconstructed into sources (src) and targets (tgt) using:

$$\mathrm{src} : (\{n \in \mathbb{N} \mid 1 \leq n \leq ||p||\} \to \mathbf{T})_{p \in \mathbb{P}} \qquad \mathrm{tgt} : \left(\{n \in \mathbb{N}^0 \mid 0 \leq n \leq ||p||\} \to \mathbb{P}\right)_{p \in \mathbb{P}}$$

$$\mathrm{src}_p(k) = \begin{cases} a & \text{if } k = 1 \text{ and } p = \mathsf{a} \to p_1 \\ \mathrm{src}_{p_1}(k-1) & \text{if } k > 0 \text{ and } p = \mathsf{a} \to p_1 \end{cases} \qquad \mathrm{tgt}_p(k) = \begin{cases} p & \text{if } k = 0 \\ \mathrm{tgt}_{p_1}(k-1) & \text{if } k > 0 \text{ and } p = \mathsf{a} \to p_1 \end{cases}$$

Types always can be decomposed into paths, which we prove in the following organization lemma.

Lemma 2.7 (Organization). *For every type a we can compute $n \in \mathbb{N}^0$ and paths $p_1, \dots, p_n \in \mathbb{P}$, such that $\bigcap_{i=1}^{n} p_i \leq \mathsf{a} \leq \bigcap_{i=1}^{n} p_i$ where multiple intersections associate to the right and $\bigcap_{i=1}^{0} p_i = \omega$.*

The beta-soundness lemma (Düdder et al. 2012) can be used to control subtyping of paths and prove the following subtype lemma for paths.

Lemma 2.8 (Path-Subtyping). *For all $n \in \mathbb{N}^0$ and $p, p_1, \dots, p_n \in \mathbb{P}$, if $\bigcap_{i=1}^{n} p_i \leq p$, then we can compute $k \in \{1, \dots, n\}$ for which $p_k \leq p$.*

The lemma on path subtyping gives rise to an algorithm deciding the subtype relation.

Lemma 2.9 (Subtype decidability). *Subtyping is decidable, i.e. there is a computable function $\leq_{\mathrm{dec}}$: $\mathbf{T} \times \mathbf{T} \to \{1, 0\}$, such that $\leq_{\mathrm{dec}} (\mathsf{a}, \mathsf{b}) = 1$ iff $\mathsf{a} \leq \mathsf{b}$.*

Lemma 2.8 is also crucial for proving an extended version of the path-lemma (Düdder et al. 2012).

$n \in \mathbb{N}^0$	Natural number or zero
$x, y, z \in \mathbb{O}$	Combinators
$\alpha, \beta, \gamma \in \mathbb{V}$	Type Variables
$C, C_1, C_2, \dots, C_n \in \mathbb{C}$	Constructor Symbols
$\lvert \cdot \rvert : \mathbb{C} \to \mathbb{N}^0$	Constructor Arity
$\leq_{\mathbb{C}} \subseteq \{(C_1, C_2) \in \mathbb{C} \times \mathbb{C} \mid \lvert C_1 \rvert = \lvert C_2 \rvert\}$	Constructor Preorder
$S : \mathbb{V} \to \mathbf{T}$	Substitution
$S^*(\sigma) =$	S lifted to $\mathbb{T}_{\mathbb{V}} \to \mathbf{T}$
$C(S^*(\sigma_1), S^*(\sigma_2), \dots, S^*(\sigma_{\lvert C \rvert}))$	if $\tau = C(\sigma_1, \sigma_2, \dots, \sigma_{\lvert C \rvert})$
ω	if $\tau = \omega$
$S^*(\sigma) \to S^*(\tau)$	if $\tau = \sigma \to \tau$
$S^*(\sigma) \cap S^*(\tau)$	if $\tau = \sigma \cap \tau$
$S(\alpha)$	if $\tau = \alpha$

$\mathbb{T}_{\mathbb{V}} \ni \sigma, \tau, \rho, \sigma_1, \sigma_2, \dots, \sigma_n ::=$	Type Scheme
$C(\sigma_1, \sigma_2, \dots, \sigma_{\lvert C \rvert})$	constant
ω	omega
$\sigma \to \tau$	function
$\sigma \cap \tau$	intersection
α	variable of $\mathbb{V}$
$\mathbf{T} = \mathbb{T}_{\emptyset} \ni \begin{matrix} \mathsf{a}, \mathsf{a}_1, \mathsf{a}_2, \dots, \mathsf{a}_n \\ \mathsf{b}, \mathsf{b}_1, \mathsf{b}_2, \dots, \mathsf{b}_n \end{matrix}$	Type
$\mathcal{T} \ni M, N, N_1, N_2, \dots, N_n ::=$	Term
x	variable
$(M\ N)$	application
$\Gamma : \mathbb{O} \to \mathbb{T}_{\mathbb{V}}$	Type Context
$\mathrm{WF} \subseteq \mathbb{V} \to \mathbf{T}$	Function space of well-formed substitutions

Subtyping ($\leq \subseteq \mathbf{T} \times \mathbf{T}$):

$$\frac{C_1 \leq_{\mathbb{C}} C_2 \qquad \lvert C_1 \rvert = \lvert C_2 \rvert \qquad \forall n : \mathsf{a}_n \leq \mathsf{b}_n}{C_1(\mathsf{a}_1, \mathsf{a}_2, \dots, \mathsf{a}_{\lvert C_1 \rvert}) \leq C_2(\mathsf{b}_1, \mathsf{b}_2, \dots, \mathsf{b}_{\lvert C_2 \rvert})}\ (\text{CAx}) \qquad \frac{}{\mathsf{a} \leq \omega}\ (\leq \omega)$$

$$\frac{}{C_1(\mathsf{a}_1, \mathsf{a}_2, \dots, \mathsf{a}_{\lvert C_1 \rvert}) \cap C_1(\mathsf{b}_1, \mathsf{b}_2, \dots, \mathsf{b}_{\lvert C_1 \rvert}) \leq C_1(\mathsf{a}_1 \cap \mathsf{b}_1, \mathsf{a}_2 \cap \mathsf{b}_2, \dots, \mathsf{a}_{\lvert C_1 \rvert} \cap \mathsf{b}_{\lvert C_1 \rvert})}\ (\text{CDist})$$

$$\frac{}{\omega \leq \omega \to \omega}\ (\to \omega) \qquad \frac{\mathsf{a}_2 \leq \mathsf{a}_1 \qquad \mathsf{b}_1 \leq \mathsf{b}_2}{\mathsf{a}_1 \to \mathsf{b}_1 \leq \mathsf{a}_2 \to \mathsf{b}_2}\ (\text{Sub}) \qquad \frac{}{(\mathsf{a} \to \mathsf{b}_1) \cap (\mathsf{a} \to \mathsf{b}_2) \leq \mathsf{a} \to \mathsf{b}_1 \cap \mathsf{b}_2}\ (\text{Dist})$$

$$\frac{}{\mathsf{a} \leq \mathsf{a} \cap \mathsf{a}}\ (\text{Idem}) \quad \frac{\mathsf{a}_1 \leq \mathsf{a}_2 \qquad \mathsf{a}_2 \leq \mathsf{a}_3}{\mathsf{a}_1 \leq \mathsf{a}_3}\ (\text{Trans}) \quad \frac{\mathsf{a} \leq \mathsf{b}_1 \qquad \mathsf{a} \leq \mathsf{b}_2}{\mathsf{a} \leq \mathsf{b}_1 \cap \mathsf{b}_2}\ (\text{GLB}) \quad \frac{}{\mathsf{a} \cap \mathsf{b} \leq \mathsf{a}}\ (\text{LUB}_1) \quad \frac{}{\mathsf{a} \cap \mathsf{b} \leq \mathsf{b}}\ (\text{LUB}_2)$$

Type assignment ($\vdash \subseteq (\mathbb{O} \to \mathbb{T}_{\mathbb{V}}) \times \mathcal{T} \times \mathbf{T}$):

$$\frac{\Gamma(x) = \tau \qquad S \in \mathrm{WF}}{\Gamma \vdash x : S^*(\tau)}\ (\text{Var}) \qquad \frac{\Gamma \vdash M : \mathsf{a} \to \mathsf{b} \qquad \Gamma \vdash N : \mathsf{a}}{\Gamma \vdash (M\ N) : \mathsf{b}}\ (\to \text{E})$$

$$\frac{\Gamma \vdash M : \mathsf{a} \qquad \mathsf{a} \leq \mathsf{b}}{\Gamma \vdash M : \mathsf{b}}\ (\leq) \qquad \frac{\Gamma \vdash M : \mathsf{a} \qquad \Gamma \vdash M : \mathsf{b}}{\Gamma \vdash M : \mathsf{a} \cap \mathsf{b}}\ (\cap \text{I})$$

Fig. 2. Type assignment rules for combinatory logic with intersection types and distributing covariant constructors.

Lemma 2.10 (Extended-Path). *For all $c \in \mathbb{O}$, and $n, m \in \mathbb{N}^0$, $N_1, \dots, N_n \in \mathcal{T}$, $p_1, \dots, p_m \in \mathbb{P}$, we have*

$$\Gamma \vdash (\dots((cN_1)N_2)\dots N_n) : \bigcap_{i=1}^{m} p_i$$

if and only if for all $k \in \{1, \dots, m\}$, there exists a well-formed substitution $S \in WF$, such that for $m' \in \mathbb{N}^0$ and $p_{m+1}, \dots p_{m+m'} \in \mathbb{P}$ with $S^*(\Gamma(c)) \leq \bigcap_{i=1}^{m'} p_{m+i}$ as obtained by the organization-lemma, there exists an $i \in \{1, \dots, m'\}$ for which $\mathrm{tgt}_{p_{m+i}}(n) \leq \mathrm{tgt}_{p_k}(n)$ and for all $j \in \{1, \dots, n\}$, we have $\Gamma \vdash N_j : \mathrm{src}_{p_{m+i}}(j)$.

The existential quantification of substitutions in the extended path-lemma can be turned into an algorithm that computes these substitutions. This algorithm will later be essential in defining a coalgebra.

Lemma 2.11 (Computational-Path). *For an arbitrary fixed Γ, given:*

1. *a computable function $WF_{dec} : (\mathbb{V} \to \mathbf{T}) \to \{0, 1\}$ deciding well-formedness ($WF_{dec}(S) = 1$ iff $S \in WF$)*
2. *a computable function $\vdash_{dec} : \mathcal{T} \times \mathbf{T} \to \{0, 1\}$ deciding type-checking ($\vdash_{dec} (M, \mathsf{a}) = 1$ iff $\Gamma \vdash M : \mathsf{a}$)*
3. *the space WF of well-formed substitutions is countable*

Context	Substitutions
$\mathbb{C} = \{A, B, C\},\ \mathbb{V} = \{\alpha\}$	$S_1 = \{\alpha \mapsto A\}, S_2 = \{\alpha \mapsto B\}, S_3 = \{\alpha \mapsto A \cap B\}$
$\lvert A\rvert = \lvert B\rvert = \lvert C\rvert = 0$	$\mathrm{WF} = \{S_1, S_2, S_3\}$
$\Gamma = \{x \mapsto A \cap B, f \mapsto \alpha \to C\}$	

$$\text{For } i, j \in 1, 2, 3:\quad \dfrac{\dfrac{\Gamma(f) = \alpha \to C \qquad S_i \in \mathrm{WF}}{\Gamma \vdash f : S_i^*(\alpha \to C)}\,(\textsc{Var}) \qquad \dfrac{\dfrac{\Gamma(x) = A \cap B \qquad S_j \in \mathrm{WF}}{\Gamma \vdash x : A \cap B}\,(\textsc{Var}) \qquad A \cap B \leq S_i(\alpha)}{\Gamma \vdash x : S_i(\alpha)}\,(\leq)}{\Gamma \vdash (f\ x) : C}\,(\to \mathrm{E})$$

Fig. 3. Examples of multiple derivation trees concluding $\Gamma \vdash (f\ x) : C$

there is an algorithm computing
$\mathrm{PathComp} : (\{(c, (N_1, \dots, N_n), (p_1, \dots, p_m)) \mid \Gamma \vdash (\dots((cN_1)N_2)\dots N_n) : \bigcap_{i=1}^{m} p_i\} \to \{1, \dots, m\} \to \mathit{WF})_{n,m \in \mathbb{N}^0}$

Given $m, n \in \mathbb{N}^0, k \in \{1, \dots m\}$ *and* $x = (c, (N_1, \dots, N_n), (p_1, \dots, p_m)) \in \mathbb{O} \times \mathcal{T}^n \times \mathbb{P}^m$,
such that $\Gamma \vdash (\dots((cN_1)N_2)\dots N_n) : \bigcap_{i=1}^{m} p_i$ *for* $m' \in \mathbb{N}^0, p_{m+1}, \dots p_{m+m'} \in \mathbb{P}$
with $\left(\mathrm{PathComp}_{(m,n)}(x)(k)\right)^* (\Gamma(c)) \leq \bigcap_{i=1}^{m'} p_{m+i}$ *as obtained by the organization-lemma,*
there exists an $i \in \{1, \dots, m'\}$ *for which* $\mathrm{tgt}_{p_{m+i}}(n) \leq \mathrm{tgt}_{p_k}(n)$ *and for all* $j \in \{1, \dots, n\}$, *we have* $\Gamma \vdash N_j : \mathrm{src}_{p_{m+i}}(j)$.

The algorithm for PathComp is obtained using Hilbert's ϵ-operator (Leino 2015) and the preconditions ensure finite runtime as well as applicability of the existing implementation in the Coq Standard Library (Bertot and Castéran 2004; The Coq development team 2016). Note that decidability of unification for intersection types is presently unknown (Dudenhefner et al. 2016), so there is no direct way to construct the result of PathComp.

Example 2.12 (Application of the path lemma). Let us explore an example (Fig. 3) of an application of the (computational) path lemma. For constructors A, B, C and type variable α, fix a context $\Gamma = \{x \mapsto A \cap B, f \mapsto \alpha \to C\}$. Define all constructors to be nullary and omit empty parentheses after their names. Choose $S_1 = \{\alpha \mapsto A\}, S_2 = \{\alpha \mapsto B\}, S_3 = \{\alpha \mapsto A \cap B\}$ to be all well-formed substitutions. The lower half of Fig. 3 depicts some of the possible derivation trees to conclude $\Gamma \vdash (f\ x) : C$. Substitutions S_i and S_j in leaf positions of the tree can be chosen arbitrarily. The extended path lemma states that there exists a substitution S and a path p, such that $S^*(\Gamma(f)) \leq p$. It asserts further that $\mathrm{tgt}_p(1) \leq C$ and $\Gamma \vdash x : \mathrm{src}_p(1)$. The computational path lemma in this case is invoked with $\mathrm{PathComp}_{(1,1)}(f, x, C)(1)$. It chooses one $S \in \mathrm{WF}$. In the proof tree, S is substitution S_i. There is no unique way to compute S, because S_1, S_2 and S_3 are all possible results, but not related by subsumption since $S_3^*(\alpha \to C) = A \cap B \to C \not\leq A \to C = S_1^*(\alpha \to C)$ and $A \to C \not\leq B \to C = S_2^*(\alpha \to C)$. We resort to trying out all possible substitutions until we find a sufficient one. Hilbert's ϵ-operator is a mathematical abstraction of this search process. By the (non-computational) path-lemma we know that S must exist and therefore it will be found at some point during the enumeration of WF. Side condition 2 of the computational path lemma ensures we can test any enumerated substitution for being sufficient and thus know when to stop searching.

Our next framework component is type inhabitation for combinatory logic, which is undecidable in its most general form (Düdder et al. 2012). However, for all finite substitution spaces we may satisfy preconditions 1 and thereby 2 of the computational path lemma, and also achieve decidable type checking and get an enumerative type inhabitation algorithm.

Theorem 2.13 (Finite Substitution Spaces). *Whenever the function space WF is finite, i.e., there is an injective map from all well-formed substitutions to a finite set of natural numbers,*

(1) *The type checking function* $\vdash_{dec}: \mathcal{T} \times \mathbf{T} \to \{0, 1\}$ *with* $(\vdash_{dec} (M, \mathsf{a}) = 1$ *iff* $\Gamma \vdash M : \mathsf{a})$ *is computable.*
(2) *Each projection of the type inhabitation set inhs:* $\left(\sum_{n \in \mathbb{N}^0 \cup \infty} \prod_{m=1}^{n} \{M \in \mathcal{T} \mid \Gamma \vdash M : \mathsf{a}\}\right)_{(\Gamma, \mathsf{a}) \in (\mathbb{O} \to \mathbb{T}_{\mathbb{V}}) \times \mathbf{T}}$ *is computable. We have* $\Gamma \vdash M : \mathsf{a}$ *iff there exists* m *such that* $m \leq \pi(\mathit{inhs}_{(\Gamma, \mathsf{a})})$ *and* $\pi_m(\pi'(\mathit{inhs}_{(\Gamma, \mathsf{a})})) = M$.

The type of inhs deserves an explanation: Given a context Γ and a closed type a, inhs returns a number n (indicating emptiness, a finite cardinality or infiniteness) and a possibly infinite sequence of terms which are inhabitants of type a. For the example in Fig. 3, $\mathsf{inhs}_{(\Gamma,C)} = (1, (f\,x))$, constructing the inhabitant $(f\,x)$ just from the specification and target type C. Testing if $n = \pi(\mathsf{inhs}_{(\Gamma,a)})$ is zero is equal to solving a type inhabitation problem (specifically, relativized type inhabitation (Düdder et al. 2012)). Algorithms for both parts of Thm. 2.13 enumerate all possible substitutions to construct minimal types (with regard to subtyping) for combinators using rules (Var) and ($\cap$I). For $\vdash_{dec}$ possible intersections of targets of the organized minimal type of the combinator in head position are checked to be subtypes of the requested type. Argument terms are checked recursively using intersections of source types. An enumerative type inhabitation algorithm inhs can be constructed based on the alternating Turing machine presented in (Düdder et al. 2012) which records traces of the machine in a tree grammar. Its nonterminals are inhabitation target types and words with parse trees rooted in those nonterminals are inhabitants. Right-hand sides of rules start with exactly one combinator symbol followed by the types of arguments required to obtain the type on the left hand side. Choices in existential quantification over recursive inhabitation targets are turned into multiple rules with the same left hand side. Termination is guaranteed by size limitation of the full tree grammar formed over nonterminals from all intersected targets of all paths of the organized minimal combinator types. For the previous example we get $A ::= x; B ::= x; A \cap B ::= x; C ::= f A \mid f B \mid f A \cap B; A \to C ::= f; B \to C ::= f; A \cap B \to C ::= f$ as the full tree grammar and a subset of these rules is generated when inhabiting A, B or C. Burghardt (2002) has proven tree grammar enumeration and finiteness checking to be feasible for the enumeration of Σ-Algebras over closed sorts without subsorting.

Remember, the reason we use intersection types is to have an ephemeral layer of semantic types on top of the specification, which is used to control composition (e.g. for eliminating semantically unwanted solutions). In order to separate these layers, we need a clear notion of disjointness.

Definition 2.14 (Disjointness).

- Disjoint sets of type schemes $T_1 \subsetneq \mathbb{T}_{\mathbb{V}_1}$ and $T_2 \subsetneq \mathbb{T}_{\mathbb{V}_2}$ are formed over disjoint variable sets $\mathbb{V}_1 \uplus \mathbb{V}_2 = \mathbb{V}$ and disjoint constructor symbol sets $\mathbb{C}_1 \uplus \mathbb{C}_2 = \mathbb{C}$ for which there are no $C_1 \in \mathbb{C}_1$ and $C_2 \in \mathbb{C}_2$ such that $C_1 \leq_{\mathbb{C}} C_2$ or $C_2 \leq_{\mathbb{C}} C_1$. This generalizes to disjoint closed types, which are subsets of $\mathbb{T}_{\emptyset} = \mathbf{T}$.
- Disjoint contexts $\Gamma_1 : \mathbb{O} \to T_1$ and $\Gamma_2 : \mathbb{O} \to T_2$ map to disjoint type schemes.
- Disjoint substitutions $S_1 : \mathbb{V}_1 \to T_1$ and $S_2 : \mathbb{V}_2 \to T_2$ map disjoint variable sets to disjoint closed types.
- Disjoint well-formedness spaces $\mathrm{WF}_1 \subset \mathbb{V}_1 \to T_1$ and $\mathrm{WF}_2 \subset \mathbb{V}_2 \to T_2$ ensure all substitutions $S_1 \in \mathrm{WF}_1$ and $S_2 \in \mathrm{WF}_2$ are disjoint. Their pairwise disjoint union $\mathrm{WF} = \bigcup_{(S_1,S_2)\in WF_1 \times WF_2} \{S_1 \uplus S_2\}$ is a decomposable well-formedness space.

Using disjointness we can freely compose and decompose type derivations.

Lemma 2.15 (Type derivation (de-)composition). *For any decomposable well-formedness space $WF = WF_1 \uplus WF_2$ with $WF_1 \subset \mathbb{V}_1 \to T_1$ and $WF_2 \subset \mathbb{V}_2 \to T_2$, disjoint contexts $\Gamma_1 : \mathbb{O} \to T_1$ and $\Gamma_2 : \mathbb{O} \to T_2$, terms M and disjoint closed types $\mathsf{a}_1 \in T_1$ and $\mathsf{a}_2 \in T_2$, we obtain: $\Gamma_1 \cap \Gamma_2 \vdash M : \mathsf{a}_1 \cap \mathsf{a}_2$ iff $\Gamma_1 \vdash M : \mathsf{a}_1$ and $\Gamma_2 \vdash M : \mathsf{a}_2$ with $\Gamma_1 \cap \Gamma_2(x) = \Gamma_1(x) \cap \Gamma_2(x)$*

For composition we show by induction on the derivation: for all $i \in \{1, 2\}$, if $\Gamma_i \vdash M : \mathsf{a}_i$ then $\Gamma \vdash M : \mathsf{a}_i$. Rule ($\cap$I) can be used to combine the separate proofs of types a_i. Decomposition can be proven by organization into disjoint paths and separate derivation of proofs for each path in each context. Disjointness and (de-)composition can be generalized to any finite number of contexts and types.

2.3 Algebra

The plan is to enumerate sort indexed carriers using type inhabitation, so types and sorts have to be related.

Definition 2.16 (Sort embedding). Fix sorts $\mathbb{S}$ and type constructors $\mathbb{C}$ preordered by $\leq_{\mathbb{S}}$ and $\leq_{\mathbb{C}}$, as well as a variable set $\mathbb{V}$ and a substitution map $\_^{+} : (\mathbb{V} \to \mathbb{S}(\emptyset)) \to \mathbb{S}(\mathbb{V}) \to \mathbb{S}(\emptyset)$. A proper sort embedding into intersection types is a variable set indexed family of functions $[\![\_]\!] : (\mathbb{S}(V) \to \mathbb{T}_V)_{V \in \{\mathbb{V}, \emptyset\}}$, such that

(1) $[\![\_]\!]$ is injective with $[\![\_]\!]^{-1} : (\mathbb{T}_V \to \mathbb{S}(V))_{V \in \{\mathbb{V}, \emptyset\}}$, i.e. for all variable sets $V \in \{\mathbb{V}, \emptyset\}$ and sorts $s \in \mathbb{S}(V)$ we have $[\![[\![s]\!]_V]\!]_V^{-1} = s$.
(2) (Un-)embeddings respect subsorting and subtyping. For all closed sorts $s_1, s_2 \in \mathbb{S}(\emptyset)$, if $s_1 \leq_{\mathbb{S}} s_2$ then $[\![s_1]\!]_\emptyset \leq [\![s_2]\!]_\emptyset$. For all $\mathsf{a}_1, \mathsf{a}_2 \in \mathbf{T}$, such that there exists $s_1, s_2 \in \mathbb{S}(\emptyset)$ with $[\![s_1]\!]_\emptyset = \mathsf{a}_1$ and $[\![s_2]\!]_\emptyset = \mathsf{a}_2$, if $\mathsf{a}_1 \leq \mathsf{a}_2$ then $[\![\mathsf{a}_1]\!]_\emptyset^{-1} \leq_{\mathbb{S}} [\![\mathsf{a}_2]\!]_\emptyset^{-1}$.
(3) Closed sorts are only embedded into paths. For all closed sorts $s \in \mathbb{S}(\emptyset)$ we have $[\![s]\!]_\emptyset \in \mathbb{P}$.
(4) (Un-)embedding distributes with lifted substitutions. For all substitutions on sorts $S \in \mathbb{V} \to \mathbb{S}(\emptyset)$ we define a substitution on types $\underline{S} : \mathbb{V} \to \mathbf{T}$ by $\underline{S}(\alpha) = [\![S(\alpha)]\!]_\emptyset$. For all open sorts $s \in \mathbb{S}(\mathbb{V})$ we require $[\![S^{+}(s)]\!]_\emptyset = \underline{S}^{*}([\![s]\!]_{\mathbb{V}})$. Similarly, for all substitutions on types $S \in \mathbb{V} \to \mathbf{T}$ we define a substitution on sorts $\overline{S} \in \mathbb{V} \to \mathbb{S}(\emptyset)$ by $\overline{S}(\alpha) = [\![S(\alpha)]\!]_\emptyset^{-1}$. For all type schemes $\sigma \in \mathbb{T}_{\mathbb{V}}$, such that there exists $s \in \mathbb{S}(\mathbb{V})$ with $[\![s]\!]_{\mathbb{V}} = \sigma$, we require $[\![\overline{S}^{+}([\![s]\!]_{\mathbb{V}}^{-1})]\!]_\emptyset = S^{*}(\sigma)$.

Whenever no confusion can arise, we drop indexes V and write $[\![\_]\!]$ or $[\![\_]\!]^{-1}$ instead of $[\![\_]\!]_V^{-1}$ or $[\![\_]\!]_V$.

Let us consider two important classes of embeddable sort structures, constants and variance labeled trees. Constants are sorts for which $\mathbb{V}$ coincides with the empty set, i.e. they are always closed. We obtain a (trivially) proper sort embedding of constants by identically mapping them to nullary constructors. In mathematical terms $\mathbb{C} = \mathbb{S}(\emptyset)$, $\leq_{\mathbb{S}} = \leq_{\mathbb{C}}$ and for all $s \in \mathbb{C}$ we have $|s| = 0$ as well as $[\![s]\!] = s()$ and $[\![s()]\!]^{-1} = s$.

Definition 2.17 (Variance Labeled Trees). Fix a countable label set $\mathbb{L}$, an arity function $(\!| \cdot |\!) : \mathbb{L} \to \mathbb{N}^0$, a pre-order on tree labels of equal arity $\leq_{\mathbb{L}} \subset \mathbb{L} \times \mathbb{L}$ and a family of variance labels $v : \left(\prod_{i=1}^{(\!|l|\!)} \{+, -, \pm\}\right)_{l \in \mathbb{L}}$

- Variance labeled trees over variables $\alpha \in \mathbb{V}$ and labels $l \in \mathbb{L}$ are elements of the set

$$\mathfrak{T}_{\mathbb{V}} \ni t_1, t_2 \dots, t_n, t_1', t_2', \dots t_n' ::= l(t_1, \dots, t_{(\!|l|\!)}) \mid \alpha$$

- Substitutions $S \in \mathbb{V} \to \mathfrak{T}_\emptyset$ can be lifted by $S^{+} = \{\alpha \mapsto S(\alpha), l(t_1, \dots, t_{(\!|l|\!)}) \mapsto l(S^{+}(t_1), \dots, S^{+}(t_{(\!|l|\!)}))\}$
- Closed variance labeled trees are pre-ordered by $\leq_{\mathfrak{T}} \subseteq \mathfrak{T}_\emptyset \times \mathfrak{T}_\emptyset$, satisfying exactly the judgmental rule

$$\frac{l_1 \leq_{\mathbb{L}} l_2 \qquad (\!|l_1|\!) = (\!|l_2|\!) \qquad \forall n : v_{l_1}(n) = v_{l_2}(n) \wedge (t_n, t_n') \in \lesseqgtr (\pi_n(v_{l_1}))}{l_1(t_1, \dots, t_n) \leq_{\mathfrak{T}} l_2(t_1', \dots, t_n')} \ (\leq_{\mathfrak{T}})$$

with $\lesseqgtr = \{+ \mapsto \leq_{\mathfrak{T}}, - \mapsto \{(t_1, t_2) \in \mathfrak{T} \times \mathfrak{T} \mid t_2 \leq_{\mathfrak{T}} t_1\}, \pm \mapsto \{(t_1, t_2) \in \mathfrak{T} \times \mathfrak{T} \mid t_1 \leq_{\mathfrak{T}} t_2 \wedge t_2 \leq_{\mathfrak{T}} t_1\}\}$.
Reflexivity, transitivity and decidability of $\leq_{\mathbb{L}}$ each imply the same property of $\leq_{\mathfrak{T}}$

Variance labeled trees are a generalization of covariant type constructors: variance labeling with v allows to control if individual children are treated as co- (+), contra- (−) or invariant (±) when comparing trees. Suppose we want to encode a sort for function arrows "$\alpha \to \beta$". Then a possible tree representation is $\to (\alpha, \beta)$ with $(\!|\to|\!) = 2$. Without subsorting we can use $v_{\to} = (\pm, \pm)$ and with subsorting $v_{\to} = (-, +)$. Contravariance and the path condition (Def. 2.16.3) prevent us from embedding variance labeled trees just by reusing labels as intersection type-constructors. In order to avoid these problems, we pick a fresh label $\bullet \notin \mathbb{L}$ and choose $\mathbb{C} = \mathbb{L} \uplus \{\bullet\}$ with $|\cdot| = (\!| \cdot |\!) \uplus \{\bullet \mapsto 0\}$ as well as $\leq_{\mathbb{C}} = \leq_{\mathbb{L}} \uplus \{(\bullet, \bullet)\}$. For $\alpha \in \mathbb{V}$, $l \in \mathbb{L}$ and $t_1, \dots, t_n \in \mathfrak{T}_{\mathbb{V}}$ we define the embedding

$$\llbracket\_\rrbracket = \{\alpha \mapsto \alpha, l(t_1, \dots, t_{\llparenthesis l \rrparenthesis}) \mapsto (l(\llbracket t_1 \rrbracket_{+-}(\pi_1(v_l)), \dots, \llbracket t_{\llparenthesis l \rrparenthesis} \rrbracket_{+-}(\pi_{\llparenthesis l \rrparenthesis}(v_l)))) \to \bullet) \to \bullet\}$$
with $\llbracket t \rrbracket_{+-} = \{+ \mapsto (\llbracket t \rrbracket \to \bullet) \to \bullet, - \mapsto (\bullet \to \llbracket t \rrbracket) \to \bullet, \pm \mapsto (\llbracket t \rrbracket \to \llbracket t \rrbracket) \to \bullet\}$ and its inverse
$$\llbracket\_\rrbracket^{-1} = \{\alpha \mapsto \alpha, (l(\sigma_1, \dots, \sigma_{\llparenthesis l \rrparenthesis}) \to \bullet) \to \bullet \mapsto l(\llbracket \sigma_1 \rrbracket_{+-}^{-1}, \dots, \llbracket \sigma_{\llparenthesis l \rrparenthesis} \rrbracket_{+-}^{-1})\} \uplus \{\sigma \mapsto t\}$$
with $\llbracket\_\rrbracket_{+-}^{-1} = \{(\sigma \to \bullet) \to \bullet \mapsto \llbracket \sigma \rrbracket^{-1}\} \uplus \{(\bullet \to \sigma) \to \bullet \mapsto \llbracket \sigma \rrbracket^{-1}\} \uplus \{(\sigma \to \sigma) \to \bullet \mapsto \llbracket \sigma \rrbracket^{-1}\} \uplus \{\sigma \mapsto t\}$
for an arbitrarily chosen fixed $t \in \mathfrak{T}_{\mathbb{V}}$.

Tree embedding $\llbracket\_\rrbracket$ is a proper sort embedding, where injectivity (Def. 2.16.1) and distribution (Def. 2.16.4) can be shown by induction on the definition of trees. The previously problematic path condition (Def. 2.16.3) always holds, because results of $\llbracket\_\rrbracket$ end in the nullary constructor $\bullet \in \mathbb{P}_{\mathbb{C}} \subset \mathbb{P}$. Subsorting and subtyping match on root nodes of trees, because root labels l are in contra-contravariant = covariant position. According to their variance labeling, $\llbracket\_\rrbracket_{+-}$ places child nodes in contra-contra = covariant, co-contravariant = contravariant and contra-contra + co-contra = invariant position. Hence by induction on the size of embedded trees, $\llbracket\_\rrbracket$ and its inverse also respect subsorting and subtyping (Def. 2.16.2).

Equipped with a notion of proper sort embeddings, we can proceed to specify a map between signature terms and terms in combinatory logic. For each given signature $\Sigma_{\mathbb{S},\mathbb{V}}$ this map is expressible as a subsorted $\Sigma_{\mathbb{S},\mathbb{V}}$-Algebra and its inverse is a subsorted $\Sigma_{\mathbb{S},\mathbb{V}}$-Coalgebra.

Theorem 2.18 (Subsorted $\Sigma_{\mathbb{S},\mathbb{V}}$-(Co-)Algebras for combinatory logic with intersection types). *Fix sorts $\mathbb{S}$ and type constructors $\mathbb{C}$ preordered by $\leq_{\mathbb{S}}$ and $\leq_{\mathbb{C}}$, as well as a finite variable set $\mathbb{V}$, a substitution map $\_^{+}$: $(\mathbb{V} \to \mathbb{S}(\emptyset)) \to \mathbb{S}(\mathbb{V}) \to \mathbb{S}(\emptyset)$ a proper sort embedding $\llbracket\_\rrbracket : (\mathbb{S}(V) \to \mathbb{T}_V)_{V \in \{\mathbb{V}, \emptyset\}}$ and a signature $\Sigma_{\mathbb{S},\mathbb{V}} = (\mathbb{O}, arity, domain, range)$. Without loss of generality require $\blacksquare \in \mathbb{C}$ to be a unary constructor symbol, i.e. $|\blacksquare| = 1$. Further, fix any well-formed substitution spaces on sorts $WF_{\mathbb{S}} \subseteq \mathbb{V} \to \mathbb{S}(\emptyset)$ and on types $WF_{\mathbf{T}} \subseteq \mathbb{V} \to \mathbf{T}$ such that $S \in WF_{\mathbb{S}}$ implies $\underline{S} \in WF_{\mathbf{T}}$, $S \in WF_{\mathbf{T}}$ implies $\overline{S} \in WF_{\mathbb{S}}$ and for all $S \in WF_{\mathbf{T}}$ and $\alpha \in \mathbb{V}$ there exists $s \in \mathbb{S}(\emptyset)$ with $S(\alpha) = \llbracket s \rrbracket$. Define a context $\Gamma : \mathbb{O} \to \mathbb{T}_{\mathbb{V}}$ by*
$$\Gamma(o) = \blacksquare(\llbracket \pi_1(domain(o)) \rrbracket) \to \blacksquare(\llbracket \pi_2(domain(o)) \rrbracket) \to \cdots \to \blacksquare(\llbracket \pi_{arity(o)}(domain(o)) \rrbracket) \to \blacksquare(\llbracket range(o) \rrbracket)$$
and a carrier $\mathbb{C}_s = \{M \in \mathcal{T} \mid \Gamma \vdash M : \blacksquare(\llbracket s \rrbracket)\}$.

1. *Morphisms $m_s : \mathbb{F}_s(\mathbb{C}) \to \mathbb{C}_s$ with $m_s((S, o, args)) = (\dots((o\ \pi_1(args))\ \pi_2(args)) \dots \pi_{arity(o)}(args))$ constitute a $\Sigma_{\mathbb{S},\mathbb{V}}$-Algebra.*
2. *For countable well-formed type substitution spaces $WF_{\mathbf{T}}$ decidable by WF_{dec} and type checking in Γ decided by $\vdash_{dec}$ (as required by the computational path lemma 2.11) morphisms*
$$m_s^{-1} : \mathbb{C}_s \to \mathbb{F}_s(\mathbb{C})$$
$$m_s^{-1}((\dots((cN_1)N_2)\dots N_n)) = (\overline{\mathrm{PathComp}_{(n,1)}(c, (N_1, N_2, \dots, N_n), \blacksquare(\llbracket s \rrbracket))(1)}, c, (N_1, N_2, \dots, N_n))$$
constitute a $\Sigma_{\mathbb{S},\mathbb{V}}$-Coalgebra.

At first glance, the definitions of Thm. 2.18 look straightforward. We create a combinator for each operation. Carrier sets include just the terms typeable by an embedded sort s. We apply or unapply combinators and some arguments to get the algebra or coalgebra. Substitutions for the coalgebra are guessed by the computational-path Lemma 2.11. A more careful second look reveals that the difficult part is to prove that morphism families m and m^{-1} really map to their indicated ranges. First, it is crucial to observe that combinators for any operation o are typeable in Γ by paths p with length arity(o), i.e., $\Gamma \vdash o : p$ and $||p|| =$ arity(o). By the distributivity (Def. 2.16.4) and path (Def. 2.16.3) side conditions of proper sort embeddings, any lifted substitution applied to $\Gamma(o)$ will yield a path p derivable by a single application of rule (Var). In the definition of Γ, embedded sorts are protected by the $\blacksquare$ type constructor. This forces the path length to be equal to the arity of o, independent of any arrows possibly introduced by the sort embedding. Note that if the embedding introduces no arrows, ■ can be omitted. In particular, this is the case for the aforementioned trivial constant embedding of closed sorts. In any case, we enforce that for the morphism family m, after arity(o) iterated applications of rule ($\rightarrow$ E) for each argument, we get the desired type and conclude that the range of m matches its definition. For the coalgebra morphisms m^{-1}, the combinator c in head position of the argument term $(\dots((cN_1)N_2)\dots N_n)$ has to be an operation because there are no other combinator symbols in Γ. By definition of $\mathbb{C}_s$ we have $\Gamma \vdash (\dots((cN_1)N_2)\dots N_n) : \blacksquare([\![s]\!])$, where $\blacksquare([\![s]\!])$ is a single path as ensured by the path (Def. 2.16.3). Thus, PathComp results in a single substitution on types satisfying the conclusions of the extended path lemma 2.10. Unembedding this substitution respects subsorting and so domain and range of c are compatible with the requirements imposed by $\mathbb{F}_s(\mathbb{C})$. Finally, arguments N_i are typable by subtypes of the path sources, so after one application of rule ($\leq$) for each argument we may see that the triple returned by m_s^{-1} is indeed an element of $\mathbb{F}_s(\mathbb{C})$.

Remember that our goal is to create terms of a target language, which are elements of some carrier $\mathbb{D}$. We therefore need for any given $\mathbb{D}$ a family of functions $f_s : \mathbb{C}_s \rightarrow \mathbb{D}_s$, mapping terms in combinatory logic (inhabitation results) to target terms. Given algebra morphisms $h_s : \mathbb{F}_s(\mathbb{D}) \rightarrow \mathbb{D}_s$ interpreting a signature in the target language, we get the commuting diagram:

$$\begin{tikzcd}
\mathbb{F}_s(\mathbb{C}) \arrow[r, "\mathbb{F}(f)_s"] & \mathbb{F}_s(\mathbb{D}) \arrow[d, "h_s"] \\
\mathbb{C}_s \arrow[u, "m_s^{-1}"] \arrow[r, "f_s"] & \mathbb{D}_s
\end{tikzcd}$$

We see that f_s can be constructed as an element of the family of least fixpoints solving the recursive equation

$$f = \left(h_s \circ \mathbb{F}(f)_s \circ m_s^{-1}\right)_{s \in \mathbb{S}(\emptyset)}$$

By induction on the size of terms in $\mathbb{C}_s$, we may show that this fixpoint family exists. We can represent the universal quantification of $\mathbb{D}$ and h by dependent products: $f_\Pi : \prod_{\mathbb{D}:\mathrm{Set}\rightarrow\mathrm{Set}} \prod_{h:\prod_{s\in\mathbb{S}(\emptyset)} \mathbb{F}_s(\mathbb{D})\rightarrow\mathbb{D}_s} \prod_{s\in\mathbb{S}(\emptyset)} \mathbb{C}_s \rightarrow \mathbb{D}_s$ satisfying $\pi_s(\pi_h(\pi_\mathbb{D}(f_\Pi))) = h_s \circ \mathbb{F}(f)_s \circ m_s^{-1}$ for all possible $\mathbb{D}$, h, and s. In Coq, dependent products are lambda abstractions, so f_Π can be represented as

```
Definition f_Pi : forall (D : Sort EmptySet → Type) (h: forall s, F D s → D s) (s: Sort EmptySet), C s → D s :=
  fun D ⇒ fun h ⇒ fun s ⇒ ...
```

where `fun x ⇒ M` denotes a lambda abstraction of `M` over `x`. All target language implementation details of the signature are filled in by $\mathbb{D}$ and h, so this is an abstraction over implementation details in a strong technical sense.

Ideally, we would like f_s to be unique. There should be only one possibility to create elements of the target carrier. Unfortunately, the definition of m_s^{-1} (Thm. 2.18) makes this impossible. As we have seen in Ex. 2.12, PathComp is free to choose from multiple possible substitutions, which renders the first component of the result of m_s^{-1} non-unique. We can, however, get uniqueness up to the choice of substitution. To make this precise, we define a relation family $\widetilde{\mathbb{F}}$ equating objects of $\mathbb{F}_s(\mathbb{E})$ for some carrier $\mathbb{E}$ independent of their substitution.

Definition 2.19 (Relating algebra inputs (coalgebra results) up to substitution). For any closed sort indexed family of sets $O_{s\in\mathbb{S}(\emptyset)}$, a closed sort indexed relation family $R^O \subseteq (O_s \times O'_s)_{(s,s')\in\mathbb{S}(\emptyset)}$ is

(1) reflexive, whenever for all $s \in \mathbb{S}(\emptyset)$ and $x \in O_s$ we have $(x, x) \in R^O_{(s,s)}$
(2) transitive, whenever for all $s_1, s_2, s_3 \in \mathbb{S}(\emptyset)$ and $x \in O_{s_1}, y \in O_{s_2}, z \in O_{s_3}$ we have $(x, y) \in R^O_{(s_1,s_2)}$ and $(y, z) \in R^O_{(s_2,s_3)}$ implies $(x, z) \in R^O_{(s_1,s_3)}$
(3) symmetric, whenever for all $s_1, s_2 \in \mathbb{S}(\emptyset)$ and $x \in O_{s_1}, y \in O_{s_2}$ we have $(x, y) \in R^O_{(s_1,s_2)}$ iff $(y, x) \in R^O_{(s_2,s_1)}$
(4) a preorder family, whenever it is reflexive and transitive
(5) an equivalence family, whenever it is reflexive, transitive and symmetric

Fix any carrier $\mathbb{E}$ and closed sort indexed relation family $\sim^{\mathbb{E}} \subseteq (\mathbb{E}_s \times \mathbb{E}_{s'})_{(s,s') \in \mathbb{S}(\emptyset) \times \mathbb{S}(\emptyset)}$ acting on carrier elements. The substitution invariant relation family $\widetilde{\mathbb{F}}$ on algebra inputs (coalgebra results) is then defined as

$$\widetilde{\mathbb{F}}(\mathbb{E})(\sim^{\mathbb{E}})_{(s,s')} = \left\{ \begin{matrix} ((S_1, o, \mathrm{args}_1), (S_2, o, \mathrm{args}_2)) \\ \in \mathbb{F}_s(\mathbb{E}) \times \mathbb{F}_{s'}(\mathbb{E}) \end{matrix} \middle| \begin{matrix} \text{for all } i \in \{1, 2, \ldots, \mathrm{arity}(o)\} : \\ \pi_i(\mathrm{args}_1) \sim^{\mathbb{E}}_{(S_1^+(\pi_i(\mathrm{domain}_o)), S_2^+(\pi_i(\mathrm{domain}_o)))} \pi_i(\mathrm{args}_2) \end{matrix} \right\}$$

and individually inherits reflexivity, transitivity and symmetry from $\sim^{\mathbb{E}}$.
For the combinatory logic carrier $\mathbb{C}_s = \{M \in \mathcal{T} \mid \Gamma \vdash M : \blacksquare([\![s]\!])\}$ and equivalence family $\sim^{\mathbb{C}}_{(s,s')} = \{(M, M) \mid M \in \mathbb{C}_s \cap \mathbb{C}_{s'}\}$ we abbreviate the equivalence family $\widetilde{\mathbb{F}}(\mathbb{C})(\sim^{\mathbb{C}})$ by $\widetilde{\mathbb{F}}^{\mathbb{C}}$.

The first part of Def. 2.19 translates preorder and equivalence relations to families of relations indexed by two different sorts. We then define $\widetilde{\mathbb{F}}$, such that it relates any two objects $(S_1, o, \mathrm{args}_1) \in \mathbb{F}_s(\mathbb{E})$ and $(S_2, o, \mathrm{args}_2) \in \mathbb{F}_{s'}(\mathbb{E})$ iff their operations are identical and all of their arguments are related by some other relation family $\sim^{\mathbb{E}}$. For the carrier $\mathbb{C}_s$, we choose $\sim^{\mathbb{C}}_{(s,s')}$ equating identical combinator terms if they are typable by both $\blacksquare([\![s]\!])$ and $\blacksquare([\![s']\!])$. The relation family $\widetilde{\mathbb{F}}(\mathbb{C})(\sim^{\mathbb{C}}) = \widetilde{\mathbb{F}}^{\mathbb{C}}$ then equates $x \in \mathbb{F}_s(\mathbb{C})$ and $y \in \mathbb{F}_{s'}(\mathbb{C})$ for identical operations and arguments, ignoring substitutions as desired ($(x, y) \in \widetilde{\mathbb{F}}^{\mathbb{C}}_{(s,s')}$ implies $\pi(\pi'(x)) = \pi(\pi'(y))$ and $\pi'(\pi'(x)) = \pi'(\pi'(y))$). We may prove congruence and bisimulation properties for $\widetilde{\mathbb{F}}^{\mathbb{C}}$, $\sim^{\mathbb{C}}$, m and m^{-1}.

Lemma 2.20 (Congruence and Bisimulation modulo substitution). *Fix any carrier $\mathbb{E}$, $\Sigma_{\mathbb{S},\mathbb{V}}$-algebra h and $\Sigma_{\mathbb{S},\mathbb{V}}$-coalgebra h^{-1} on $\mathbb{E}$ together with a closed sort indexed relation family $\sim^{\mathbb{E}} \subseteq (\mathbb{E}_s \times \mathbb{E}_{s'})_{(s,s') \in \mathbb{S}(\emptyset) \times \mathbb{S}(\emptyset)}$. If for all sorts $s_1, s_2 \in \mathbb{S}(\emptyset)$ the diagram*

(1)
$$\begin{tikzcd}
\mathbb{F}_{s_1}(\mathbb{E}) \arrow[d, "h_{s_1}"] & \widetilde{\mathbb{F}}(\mathbb{E})(\sim^{\mathbb{E}})_{(s_1,s_2)} \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] \arrow[d, "h_{s_1} \times h_{s_2}"] & \mathbb{F}_{s_2}(\mathbb{E}) \arrow[d, "h_{s_2}"] \\
\mathbb{C}_{s_1} & \sim^{\mathbb{E}}_{(s_1,s_2)} \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & \mathbb{C}_{s_2}
\end{tikzcd}$$

(2)
$$\begin{tikzcd}
\mathbb{C}_{s_1} \arrow[d, "h^{-1}_{s_1}"] & \sim^{\mathbb{E}}_{(s_1,s_2)} \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] \arrow[d, "h^{-1}_{s_1} \times h^{-1}_{s_2}"] & \mathbb{C}_{s_2} \arrow[d, "h^{-1}_{s_2}"] \\
\mathbb{F}_{s_1}(\mathbb{E}) & \widetilde{\mathbb{F}}(\mathbb{E})(\sim^{\mathbb{E}})_{(s_1,s_2)} \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & \mathbb{F}_{s_2}(\mathbb{E})
\end{tikzcd}$$

(1) commutes, that is $h_{s_1} \circ \pi_1 = \pi_1 \circ (h_{s_1} \times h_{s_2})$ and $h_{s_2} \circ \pi_2 = \pi_2 \circ (h_{s_1} \times h_{s_2})$, then $\sim^{\mathbb{E}}$ is an algebra congruence relation family.

(2) commutes, that is $h^{-1}_{s_1} \circ \pi_1 = \pi_1 \circ (h^{-1}_{s_1} \times h^{-1}_{s_2})$ and $h^{-1}_{s_2} \circ \pi_2 = \pi_2 \circ (h^{-1}_{s_1} \times h^{-1}_{s_2})$, then $\sim^{\mathbb{E}}$ is a coalgebra bisimulation relation family.

Equivalence family $\sim^{\mathbb{C}}$ is an algebra congruence and a coalgebra bisimulation family for the algebra m and coalgebra m^{-1} constructed in Thm. 2.18.

The above Lem. 2.20 guarantees that algebra m and coalgebra m^{-1} from Thm. 2.18 map equivalent inputs to equivalent outputs with respect to $\sim^{\mathbb{C}}$ and $\widetilde{\mathbb{F}}^{\mathbb{C}}$. We use it to show that the constructed morphisms f into the target language are unique modulo these relations.

Lemma 2.21 (Uniqueness up to Substitution). *Fix any carrier $\mathbb{D}$, an algebra with morphisms $h_s : \mathbb{F}_s(\mathbb{D}) \longrightarrow \mathbb{D}$ and a relation family $\sim^{\mathbb{D}}$ on $\mathbb{D}$ that is transitive and a congruence relation family for h. Using m and m^{-1} from Thm. 2.18 the family of least fixpoints $f_s : \mathbb{C}_s \longrightarrow \mathbb{D}_s$ satisfying $f_s = h_s \circ \mathbb{F}(f)_s \circ m_s^{-1}$ has the following properties for all $s_1, s_2 \in \mathbb{S}(\emptyset)$, $x \in \mathbb{F}_{s_1}(\mathbb{C})$, $y \in \mathbb{F}_{s_2}(\mathbb{C})$ and $N_1 \in \mathbb{C}_{s_1}$, $N_2 \in \mathbb{C}_{s_2}$ such that $(x, y) \in \widetilde{\mathbb{F}}^{\mathbb{C}}_{(s_1,s_2)}$ and $N_1 \sim^{\mathbb{C}}_{(s_1,s_2)} N_2$:*

- *f respects the relation families: $f_{s_1}(N_1) \sim^{\mathbb{D}}_{(s_1,s_2)} f_{s_2}(N_2)$*
- *f is a family of algebra morphisms up to substitution: $f_{s_1}(m(x)) \sim^{\mathbb{C}}_{(s_1,s_2)} h_{s_2}(\mathbb{F}(f)_{s_2}(y))$*
- *f is unique modulo substitution: for all families of algebra morphisms up to substitution $g_s : \mathbb{C}_s \longrightarrow \mathbb{D}_s$, such that g also respects the relation families, we have $f_{s_1}(N_1) \sim^{\mathbb{D}}_{(s_1,s_2)} g_{s_2}(N_2)$*

Lemma 2.21 is free: it can be proven by generalization of $\mathbb{C}$, m, m^{-1} and $\sim^{\mathbb{C}}$ to carriers and algebra-coalgebra-pairs inverse modulo preorder relation families $\sim^{\mathbb{C}}$, as long as well-founded induction on carrier elements is permitted and there is a notion of getting smaller arguments after application of m^{-1}. The last bullet point of Lem. 2.21 is the most important property of our constructed translation f. It says: no matter how we choose target carriers $\mathbb{D}$ and target algebras h, whenever there is a congruence relation family $\sim^{\mathbb{D}}$ for h, our constructed translation family, f is unique up to this congruence. That is, for any inputs equal up to $\sim^{\mathbb{C}}$ and any other proper translation g, outputs of f and g are equal modulo $\sim^{\mathbb{D}}$. Here, proper means g maps $\sim^{\mathbb{C}}$-related inputs to $\sim^{\mathbb{D}}$-related outputs and it is an algebra morphism family up to substitution. Algebra morphism families make the following diagram commute for all $s \in \mathbb{S}(\emptyset)$:

$$\begin{tikzcd}
\mathbb{F}_s(\mathbb{C}) \arrow[r, "\mathbb{F}(g)_s"] \arrow[d, "m_s"] & \mathbb{F}_s(\mathbb{D}) \arrow[d, "h_s"] \\
\mathbb{C}_s \arrow[r, "g_s"] & \mathbb{D}_s
\end{tikzcd}$$

An algebra morphism family up to substitution satisfies weakened conditions on commutation. In equations this just means replacing = by the previously specified relation families: for all $s_1, s_2 \in \mathbb{S}(\emptyset)$ and $x \in \mathbb{F}_{\mathbb{C}}(s_1)$, $y \in \mathbb{F}_{\mathbb{C}}(s_2)$ such that $(x, y) \in \widetilde{\mathbb{F}}^{\mathbb{C}}_{(s_1,s_2)}$ we have $f_{s_1}(m(x)) \sim^{\mathbb{C}}_{(s_1,s_2)} h_{s_2}(\mathbb{F}(g)_{s_2}(y))$.

Before we continue, we should investigate two more properties of the constructed family of translations f. First, we would like to be sure only to create target objects composed of h-translated signature operations. Second, we would like to create all of these objects. Let us define what it means to be composed of h-translated signature operations.

Definition 2.22 (Algebraically generated objects). For any carrier $\mathbb{D} : \mathbb{S}(\emptyset) \to \text{Set}$ and an algebra with morphisms $h_s : \mathbb{F}_s(\mathbb{D}) \to \mathbb{D}$, the closed sort indexed family of $\Sigma_{\mathbb{S},\mathbb{V}}$-algebraically generated objects is defined as

$$\mathcal{AG}(\mathbb{D})(h)_s = \left\{ h_s((S, o, \text{args})) \,\middle|\, \begin{array}{l} (S, o, \text{args}) \in \mathbb{F}_s(\mathbb{D}) \text{ and} \\ \text{for all } i \in \{1, 2, \ldots, \text{arity}(o)\} : \pi_i(\text{args}) \in \mathcal{AG}(\mathbb{D})(h)_{S^+(\pi_i(\text{domain}_o))} \end{array} \right\}$$

Set $\mathcal{AG}(\mathbb{D})(h)_s$ contains exactly the objects in $\mathbb{D}$ which have a pre-image in $\mathbb{F}_s(\mathbb{D})$ under the algebra morphism h_s, for which all arguments are included in one of the $\mathcal{AG}(\mathbb{D})(h)$ sets, which is chosen according to their sort. Thus, all elements of $\mathcal{AG}(\mathbb{D})(h)_s$ can be constructed by h_s applied to signature operations and arguments which are again h-translated. This set can be inductively defined starting with pre-image operations of arity 0, which require no recursion in the definition. Allowing induction is important to show the soundness and completeness lemma for f with respect to $\mathcal{AG}$.

Lemma 2.23 (Soundness and Completeness). *Fix any carrier $\mathbb{D}$, an algebra with morphisms $h_s : \mathbb{F}_s(\mathbb{D}) \to \mathbb{D}$ and a relation family $\sim^{\mathbb{D}}$ on $\mathbb{D}$ that is transitive and a congruence relation family for h. Using the fixpoint family $f : (\mathbb{C}_s \to \mathbb{D}_s)_s$ from Lem. 2.21, one has the following properties for all $s \in \mathbb{S}(\emptyset)$:*

- *f is sound: for all $c \in \mathbb{C}_s$ we have $f_s(c) \in \mathcal{AG}(\mathbb{D})(h)_s$*
- *f is complete up to substitution: for all $d \in \mathcal{AG}(\mathbb{D})(h)_s$ there exists $c' \in \mathbb{C}_s$ such that $f_s(c') \sim^{\mathbb{D}}_{(s,s)} d$*

Lemma 2.23 is again free: to prove it we generalize just as in the proof of Lem. 2.21 with the additional restriction that $\sim^{\mathbb{D}}$ has to be an equivalence relation family. Then, the first part follows by induction on c and the definition of f. The second part is by induction on d, starting with pre-image operations of arity 0, proceeding from arity n to arity $n + 1$. Completeness is up to substitution for the same reason as uniqueness is only up to substitution: in the coalgebra m^{-1} used in f, we only pick one substitution out of the many possible. Completeness up to substitution is still strong enough to ensure that whenever there exists any algebraically generated object $d \in \mathcal{AG}(\mathbb{D})(h)_s$, we will find $c' \in \mathbb{C}_s$ translatable by f to an $\sim^{\mathbb{D}}$-equal object in $\mathbb{D}_s$. As a special case, checking emptiness of $\mathbb{C}_s$ is enough to decide emptiness of $\mathcal{AG}(\mathbb{D})(h)_s$.

$$
\begin{array}{rll}
\mathbf{T}^{\Box} \ni A, A_1, A_2, \ldots, A_n, B, C & ::= \mathrm{nat} \mid A \to B \mid A_1 \times A_2 \mid () \mid \Box A & \text{Types} \\
\mathbb{C}^{\Box} \ni M, M_1, M_2, \ldots M_n, N & ::= x \mid \lambda x.M \mid MN \mid \mathrm{box}\ M \mid \mathrm{unbox}_n\ M \mid & \text{Terms} \\
& \quad \mathrm{fix}\ M \mid \langle\rangle \mid \langle M_1, M_2\rangle \mid \mathrm{fst}\ M \mid \mathrm{snd}\ M \mid & \\
& \quad \mathrm{s} \mid \mathrm{z} \mid (\mathrm{case}\ N\ \mathrm{of}\ \mathrm{z} \Rightarrow M_1 \mid \mathrm{s}\ M_3 \Rightarrow M_4) &
\end{array}
$$

$$
\Gamma ::= \cdot \mid \Gamma, x : A \quad \text{Contexts} \qquad \Psi ::= \odot \mid \Psi; \Gamma \quad \text{Context Stacks}
$$

$$
\frac{x : A \in \Gamma}{\Psi; \Gamma \vdash^i x : A}\ (\textsc{tpi\_var}) \quad
\frac{\Psi; (\Gamma, x : A) \vdash^i M : B}{\Psi; \Gamma \vdash^i \lambda x.M : A \to B}\ (\textsc{tpi\_lam}) \quad
\frac{\Psi; \Gamma \vdash^i M : A \to B \qquad \Psi; \Gamma \vdash^i N : A}{\Psi; \Gamma \vdash^i M\ N : B}\ (\textsc{tpi\_app})
$$

$$
\frac{\Psi; \Gamma; \cdot \vdash^i M : A}{\Psi; \Gamma \vdash^i \mathrm{box}\ M : \Box A}\ (\textsc{tpi\_box}) \quad
\frac{\Psi; \Gamma \vdash^i M : \Box A}{\Psi; \Gamma; \Gamma_1; \ldots; \Gamma_n \vdash^i \mathrm{unbox}_n\ M : A}\ (\textsc{tpi\_unbox})
$$

Fig. 4. Core Rules of Mini-ML$^{\Box}$. See (Davies and Pfenning 2001) for rules for unit, product, nat and fixpoint terms.

Recall that the inhabitation set inhs is complete, so we have a sound–as well as up to substitution unique and complete enumeration of $\mathcal{AG}(\mathbb{D})(h)_s$, using $\mathbb{C}_s = \bigcup_{i=0}^{\pi(\mathsf{inhs}_{(\Gamma,\blacksquare(\llbracket s \rrbracket))})} \{\pi_i(\pi'(\mathsf{inhs}_{(\Gamma,\blacksquare(\llbracket s \rrbracket))}))\}$ and translating each element of $\mathbb{C}_s$ by f_s. Again, as a special case we know $\mathcal{AG}(\mathbb{D})(h)_s$ is empty (there is no object composed of signature operations and h-translated arguments) whenever $\pi(\mathsf{inhs}_{(\Gamma,\blacksquare(\llbracket s \rrbracket))}) = 0$. We have reached our mathematical objectives.

3 SYNTHESIS FOR MINI-ML$^{\Box}$

Davies and Pfenning (2001) present the type system Mini-ML$^{\Box}$, which allows for type safe staged computation. Mini-ML$^{\Box}$ has a modal type constructor $\Box$, such that $\Box A$ represents a value standing for code, that when executed yields a value of type A. There are operations box and unbox for translating values to and from their representation as code. On the type level, these operations introduce or remove $\Box$-constructors and the type-system is defined such that unboxing of $\Box A$ always produces type-correct terms of type A. Under the Curry-Howard-Isomorphism the type system corresponds to intuitionistic modal logic S4. Düdder et al. (2014) use a restricted version without unboxing in a synthesis approach also based on type inhabitation in combinatory logic with intersection types. In our proofs and examples we consider a curry style version of the full Mini-ML$^{\Box}$ system. The essential part of the type rules of Mini-ML$^{\Box}$ is given in Fig. 4. The first three type judgments are just those of the simply typed λ-calculus extended with a stack of contexts. Each context in the stack represents assumptions of an additional phase of staging, where the topmost stack level contains runtime assumptions of the final stage of execution. We use $\odot$ to indicate the context stack start. Rule ($\textsc{tpi\_box}$) allows to introduce a new box constructor, but only if term M contains no free program variables at the topmost staging level. This ensures that rule ($\textsc{tpi\_unbox}$) can make values typed in an earlier stage available in later stages without ever creating inconsistent assumptions. The unbox_n term constructor is indexed by a natural number to allow going back multiple levels. Type judgments for nat, pairs $A_1 \times A_2$ and unit () are standard, so we omit them for space reasons. Reduction semantics of Mini-ML$^{\Box}$ is investigated by Davies and Pfenning (2001), but we do not need to concern ourselves with it here: in the algebraic framework, terms will become carrier elements, hence reduction is just one of the arbitrarily many possible relations on these elements.

We aim to enumerate typed terms of Mini-ML$^{\Box}$, so we choose a sort structure representing types. Variance labeled trees from Def. 2.17 can be easily instantiated for this purpose. We pick $\mathbb{L} = \{\mathrm{nat}, \to, \times, (), \Box\}$, $\leq_{\mathbb{L}} = \{(l, l) \mid l \in \mathbb{L}\}$, $(\!| \cdot |\!) = \{\mathrm{nat} \mapsto 0, \times \mapsto 2, \to \mapsto 2, () \mapsto 0, \Box \mapsto 1\}$ and require $\pi_i(v_l) = \pm$ for all $l \in \mathbb{L}$ and $i \in \{1, \ldots, (\!|l|\!)\}$. For these choices it is easy to see that $\mathfrak{T}_{\emptyset} \cong \mathbf{T}^{\Box}$, because trees are just types with constructors written in prefix notation. The system could be extended to allow for subtype polymorphism by introducing more primitive types, a subtyping judgment, adjusting $\leq_{\mathbb{L}}$ and specifying co-contra variance rules for constructors, e.g. $v_{\times} = (+, +)$.

We choose closed Mini-ML$^{\Box}$ programs as target carrier $\mathbb{D}_s = \{M \mid \odot; \cdot \vdash^i M : s\}$ where $s \in \mathbb{S}(\emptyset) = \mathfrak{T}_{\emptyset}$ and we implicitly convert between $\mathbf{T}^{\Box}$ and $\mathfrak{T}_{\emptyset}$. The restriction to closed programs is helpful to be able to freely use rule (TPI_BOX). It is also reasonable, because ultimately, we are interested in executable programs that do not get stuck by evaluating assumptions from a context. We can define a signature relative to a list of implementations.

Definition 3.1 (Signatures for typed closed Mini-ML$^{\Box}$ terms). Given n terms $\vec{\mathbf{M}} = (M_1, M_2, \ldots, M_n)$ and types $\vec{\mathbf{T}} = (A_1, A_2, \ldots, A_n)$ such that $\odot; \cdot \vdash^i M_j : A_j$ for all $j \in \{1, 2, \ldots, n\}$ we define a signature $\Sigma_{\mathbb{S},\mathbb{V}}$ over sorts $\mathbb{S} = \mathfrak{T}$ as explained above and variables $\mathbb{V} = \{\alpha, \beta, \gamma\}$ by:

- op $= \{M_1, M_2, \ldots, M_n, \text{box}, \text{unbox}_0, \text{app}\}$
- arity $= \{\text{box} \mapsto 1, \text{unbox}_0 \mapsto 1, \text{app} \mapsto 2\} \uplus \{M_i \mapsto 0 \mid i \in \{1, 2, \ldots, n\}\}$
- domain $= \{\text{box} \mapsto \gamma, \text{unbox}_0 \mapsto \Box\gamma, \text{app} \mapsto (\alpha \to \beta, \alpha)\} \uplus \{M_i \mapsto () \mid i \in \{1, 2, \ldots, n\}\}$
- range $= \{\text{box} \mapsto \Box\gamma, \text{unbox}_0 \mapsto \gamma, \text{app} \mapsto \beta\} \uplus \{M_i \mapsto A_i \mid i \in \{1, 2, \ldots, n\}\}$

In shorthand notation, we can write

$$\Sigma_{\mathbb{S},\mathbb{V}} = \{M_1 : () \twoheadrightarrow A_1, M_2 : () \twoheadrightarrow A_2, \ldots, M_n : () \twoheadrightarrow A_n, \text{box} : \gamma \twoheadrightarrow \Box\gamma, \text{unbox}_0 : \Box\gamma \twoheadrightarrow \gamma, \text{app} : (\alpha \to \beta, \alpha) \twoheadrightarrow \beta\}$$

We can see that we add operations without parameters for each given implementation. Applicative composition of terms will be facilitated using the apply operation app. Further, we add unary operations for boxing and unboxing terms. To make variable instantiations finite, we restrict the space of well-formed substitutions as follows:

$$\text{WF} = \{S \in \mathbb{V} \to \mathbb{S}(\emptyset) \mid S(\alpha) = S(\beta) = S(\gamma) = () \text{ or exists } i \in \text{Nat s.t. } S(\alpha) \to S(\beta) \in \text{Tgts}_{A_i} \text{ and } S(\gamma) = ()\} \cup$$
$$\left\{ S \in \mathbb{V} \to \mathbb{S}(\emptyset) \,\middle|\, \begin{array}{l} \text{ex. } i \in \text{Nat}, n \in \mathbb{N}^0, A \in \mathbf{T}^{\Box} \text{ s.t.} \\ \quad A \in \text{Tgts}_{A_i} \cap \text{Lift}_{(n, S(\gamma))}, \text{lvl}_{\Box}(S(\gamma)) < \text{maxlvl, and } S(\alpha) = S(\beta) = () \end{array} \right\}$$

using
$$\text{Tgts}_A = \begin{cases} \{B \to C\} \cup \text{Tgts}_C & \text{if } A = B \to C \\ \{\Box B\} \cup \text{Tgts}_B & \text{if } A = \Box B \\ \{A\} & \text{otherwise} \end{cases} \qquad \text{lvl}_{\Box}(A) = \begin{cases} 1 + \text{lvl}_{\Box}(B) & \text{if } A = \Box B \\ \max\{\text{lvl}_{\Box}(B), \text{lvl}_{\Box}(C)\} & \text{if } A = B \to C \\ 0 & \text{otherwise} \end{cases}$$
$$\text{Lift}_{(n,A)} = \begin{cases} \{A\} & \text{if } n = 0 \\ \{A\} \cup \text{Lift}_{(n-1, \Box A)} & \text{otherwise} \end{cases} \qquad \text{maxlvl} = \max\{\text{lvl}_{\Box}(A_1), \text{lvl}_{\Box}(A_2), \ldots, \text{lvl}_{\Box}(A_n)\}$$

Substitutions can return (), if no substitution is required, as is the case for operations M_i. For operation app substitutions of α and β are valid, if the arrow $\alpha \to \beta$ present in target position of any implementation type A_i, possibly below a box. For operations box and unbox$_0$, substitutions of γ are any type in a target position of an implementation type, prefixed by boxes up to the maximal level of box nesting present in any implementation type. The level restriction will not cause loss of relevant solutions, because boxing types beyond the level present in implementation types just leads to trivially shifting solutions M to box M. We can use the following $\Box$-Applicative fragment of Mini-ML$^{\Box}$ relative to the given implementations to define the space of interesting solutions up to a fixed level of nested boxes.

Definition 3.2 ($\Box$-Applicative fragment of Mini-ML$^{\Box}$ implementations). Given terms $\vec{\mathbf{M}}$ and types $\vec{\mathbf{T}}$ as in Def. 3.1, we define the $\Box$-applicative fragment at type A up to level m by:

$$\text{App}^{\Box}_{(m,A)} = \{M_i \mid \odot; \cdot \vdash M_i : A \text{ and } i \in \{1, \ldots, n\}\} \cup \{M\, N \mid \text{ex. } B \in \mathbf{T}^{\Box} \text{ s.t. } M \in \text{App}^{\Box}_{(m, B \to A)} \text{ and } N \in \text{App}^{\Box}_{(m,B)}\}$$
$$\uplus \{\text{box } M \mid \text{exists } B \in \mathbf{T}^{\Box} \text{ s.t. } A = \Box B \text{ and } M \in \text{App}^{\Box}_{(m,B)}\} \cup \{\text{unbox}_0\, M \mid M \in \text{App}^{\Box}_{(m, \Box A)}\}$$

By definition we have $M \in \text{App}^{\Box}_{(m,A)}$ only if M is formed by boxing, unboxing or applying existing implementations. Further, if $M \in \text{App}^{\Box}_{(\text{maxlvl},A)}$ we can show $\odot; \cdot \vdash^i M : A$. We are now ready to define an algebra for signature Σ and the substitution space.

Theorem 3.3 (A $\Sigma_{\mathbb{S},\mathbb{V}}$-Algebra for Mini-ML$^\Box$). *Given terms $\vec{\boldsymbol{M}}$, types $\vec{\boldsymbol{T}}$ and signature $\Sigma_{\mathbb{S},\mathbb{V}}$ from Def. 3.1 as well as the previously defined well-formed space WF, we can define a $\Sigma_{\mathbb{S},\mathbb{V}}$-algebra*

$$\begin{aligned} &h : (\mathbb{F}_s(\mathbb{D}) \to \mathbb{D}_s)_{s \in \mathbb{S}(\emptyset)} \\ &h_s = \{(S, M_i, ()) \mapsto M_i \mid S \in \mathit{WF}, i \in \{1, \dots, n\}, \mathit{and}\, A_i = s\} \uplus \{(S, \mathit{app}, (M, N)) \mapsto M\ N \mid S \in \mathit{WF}\ \mathit{and}\ s = S(\beta)\} \\ &\qquad \uplus \{(S, \mathrm{box}, M) \mapsto \mathrm{box}\ M \mid S \in \mathit{WF}\ \mathit{and}\ s = \Box S(\gamma)\} \uplus \{(S, \mathrm{unbox}_0, M) \mapsto \mathrm{unbox}_0\, M \mid S \in \mathit{WF}\ \mathit{and}\ s = S(\gamma)\} \end{aligned}$$

with the following properties:

(1) h is sound and complete for the $\Box$-Applicative up to level maxlvl, *that is $\mathcal{AG}(\mathbb{D})(h)_s = \mathrm{App}^{\Box}_{(\mathrm{maxlvl}, s)}$*

(2) syntactic term equality $\equiv_{(s_1,s_2)} = \{(M, M) \mid \odot; \cdot \vdash^i M : s_1$ and $\odot; \cdot \vdash^i M : s_2\}$ is transitive and an algebra congruence for h.

Part one of Thm. 3.3 can be proven by induction on the definition of $\mathcal{AG}$, while part 2 just requires a case analysis on the operations. Note that part 2 would have been more difficult for a Church style version of Mini-ML$^\Box$: one would then have to prove that term definitions do not depend on the choice of substitutions. In our case this is also true, because we only consider closed implementation types A_i, and application, boxing and unboxing do not introduce syntactically substitution dependent binders. From the soundness and completeness lemma 2.23 and the soundness and completeness proven in Thm. 3.3 we can immediately conclude that enumerative type inhabitation $\mathsf{inhs}_{(\Gamma, \blacksquare(\llbracket s \rrbracket))}$ provides a sound and complete enumeration of the $\Box$-Applicative fragment $\mathrm{App}^{\Box}_{(\mathrm{maxlvl}, s)}$ when Γ is constructed as described in Thm. 2.18.

In our case it can be desirable to drop completeness. Suppose we have $A_i = \Box A$, then $\{M_i, \mathrm{box\ unbox}_0\, M_i,$ $\mathrm{box\ unbox}_0\ \mathrm{box\ unbox}_0\, M_i, \dots\} \subseteq \mathrm{App}^{\Box}_{(\mathrm{maxlvl}, A_i)}$, hence our solution contains infinitely many redundant terms. This can be easily avoided using semantic types. We just have to define the additional semantic type context

$$\begin{aligned} \Gamma' = &\{M_i \mapsto \mathrm{Ok}(\mathrm{ToBox}, \mathrm{ToUnbox}) \mid i \in \{1, \dots, n\}\} \uplus \{\mathrm{app} \mapsto \mathrm{Ok}(\omega, \omega) \to \mathrm{Ok}(\omega, \omega) \to \mathrm{Ok}(\mathrm{ToBox}, \mathrm{ToUnbox})\} \\ &\uplus \{\mathrm{box} \mapsto \mathrm{Ok}(\mathrm{ToBox}, \omega) \to \mathrm{Ok}(\mathrm{ToBox}, \omega), \mathrm{unbox}_0 \mapsto \mathrm{Ok}(\omega, \mathrm{ToUnbox}) \to \mathrm{Ok}(\omega, \mathrm{ToUnbox})\}. \end{aligned}$$

By the decomposition lemma 2.15 using type inhabitation requests $\mathsf{inhs}_{(\Gamma \cap \Gamma', \blacksquare(\llbracket s \rrbracket) \cap \mathrm{Ok}(\omega, \omega))}$ will produce valid solutions for s and ensure semantic soundness $\Gamma' \vdash M : \mathrm{Ok}(\omega, \omega)$. It is easy to check that the definition of Γ' prevents terms of form box $\mathrm{unbox}_0\, M$ or unbox_0 box M, since combinators box and unbox_0 require ToBox and ToUnbox from their arguments. All other combinators re-allow boxing and unboxing, so other solutions do not get lost. Semantic types are also present in the system presented in (Düdder et al. 2014), where they can be placed more freely, but do not come with a formalized notion of type constructors or semantic soundness guarantees.

4 IMPLEMENTATION AND EVALUATION

The Coq framework formalization is complete to the point where it can generate tree grammars and translate inhabitants. It uses no additional axioms, so it is executable and could be used as a baseline implementation. For practical reasons, the authors have created a second implementation in Scala. It has three major benefits. As we have seen in the introductory example, the API for creating repositories, inhabitation requests and mapping results to implementations is lightweight and does not require formal proofs or knowledge of dependent types. Further, performance is sufficient even for larger examples, such as the solitaire product line we will discuss later. The theorem prover code would need significant improvements before being competitive: to make proofs easier, it uses unary numbers and encodes powersets as lists, which has severely negative performance consequences. Finally, Scala enables access to a rich ecosystem of libraries to help implement interesting algebras. Examples for these libraries are JavaParser (van Bruggen et al. 2017) and Twirl (Vlugter et al. 2017) which are used for templating ASTs of Java programs in the solitaire example. Before we get into the details of this example, let us discuss the core components of the Scala implementation.

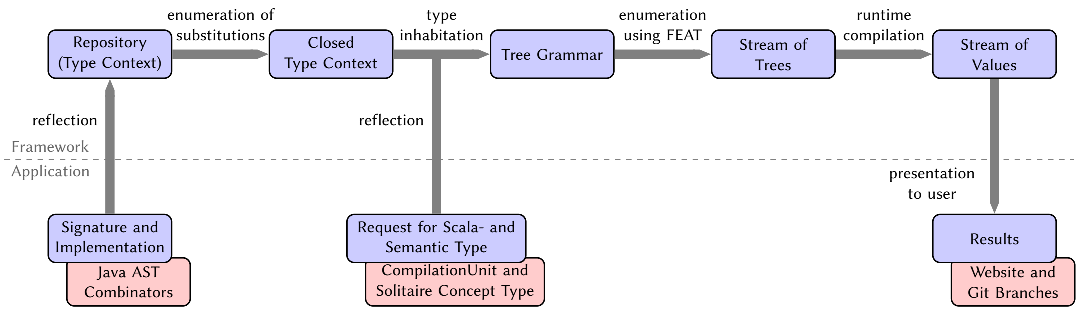

Fig. 5. Overview of the implementation data flow

```
1   sealed trait Liquid; case class White extends Liquid; case class Yolk extends Liquid
2
3   abstract class Analyst[A](specialty: Type) {
4     def apply(objects: (A, A)): A
5     val semanticType = ('Pair(Omega, specialty) =>: specialty) :&: ('Pair(specialty, Omega) =>: specialty)
6   }
7   @combinator object BiscuitPastryMaster extends Analyst[Liquid]('EggYolk) {
8     def apply(eggFilling: (Liquid, Liquid)): Liquid =
9         eggFilling match { case (x@Yolk(), _) => x; case (_, y@Yolk()) => y }
10  }
```

Listing 4. Complete code for the Biscuit Pastry Master from Listing 1

4.1 Scala Implementation

Figure 5 shows an overview of the implementation data flow. Parts above the dashed line are application independent components that can be reused as part of a framework, while parts below the dashed line are application specific. Red boxes show the application specific parts for the solitaire product line example.

We start in the lower left corner by providing a signature and its implementation. We have already seen the input language for this specification in the introduction example (Listings 1 and 2): a signature and its implementation are defined by a collection of Scala objects with `apply` methods. The object name corresponds to the signature operation. Parameters of `apply` form the operation domain and the result type of `apply` is the operation range. Implementing `apply` for an object `o` corresponds to mathematically defining the target algebra h at the operation `o`. Target carrier $\mathbb{D}_{\mathtt{A}}$ is given by the set of Scala values of type `A`. The framework extracts all necessary information from `@combinator` annotated objects using Scala's powerful reflection mechanism (Miller et al. 2017). For simplicity, the current implementation only accepts `apply` methods without generic type parameters. This means we can translate each Scala type to exactly one closed sort using the trivial constant sort embedding discussed in Sect. 2.3. We may also avoid using the additional $\blacksquare$ constructor, because the constant sort embedding introduces no arrows. Scala's reflection mechanism is also used to build the subsorting relation $\leq_{\mathbb{S}}$ as a table of all subtypes `A <: B` for types `A` and `B` occurring in the signature. Additionally, semantic types of operations can be specified by adding a field `semanticType` to the individual objects. A small embedded domain specific language allows to define intersection types in Scala syntactically close to mathematical notation: we use `:&:` for intersections $\cap$, `=>:` for arrows $\rightarrow$ and `'C('D, 'E)` for a type constructor with arguments $C(D(), E())$. Semantic types support variables, which are given as objects obtained by the constructor `Variable` with the variable name as argument. The type constant ω is simply denoted by `Omega`. Let us recapture the code of `BiscuitPastryMaster` from Listing 1 and complete it with an implementation (Listing 4). We define appropriate Scala types (line 1) and perform a pattern match on the input of the `apply` method (line 9 ff.). The resulting signature after reflection contains an operation `BiscuitPastryMaster`. The domain is a sort `(Liquid, Liquid)` and the range is `Liquid`.

```
final def inhabitStep(result: TreeGrammar, tgt: Type): (TreeGrammar, Stream[Stream[Type]]) = {
  val knownSupertypes = findSupertypeEntries(result, tgt)
  if (knownSupertypes.values.exists(Set.empty)) (removeEntriesWithArgument(result, tgt), Stream.empty)
  else findSmallerEntry(knownSupertypes, tgt) match {
    case Some(kv) => (substituteArguments(result, tgt, kv._1), Stream.empty #:: Stream.empty)
    case None =>
      val orgTgt = Organized(tgt)
      val recursiveTargets: ParMap[String, Iterable[Seq[Type]]] =
        organizedRepository.par.mapValues { cType =>
        cType.paths.flatMap(relevantFor(orgTgt, _))
          .groupBy(x => x._1.size)
          .mapValues (pathComponents => covers(orgTgt, pathComponents))
          .values.map(_.distinct).flatten
      }
      val newProductions: Set[(String, Seq[Type])] =
        recursiveTargets.toSeq.flatMap { case (c, tgts) => tgts.map((c, _)) }.toSet.seq
      val newTargets: Stream[Stream[Type]] =
        recursiveTargets.values.flatMap { tgtss => tgtss.toStream.map(tgts => tgts.toStream) }.toStream
      (result + (tgt -> newProductions), newTargets)
  } }
```

Listing 5. Inhabitation algorithm step

For the resulting type context we get

$$\Gamma(\texttt{BiscuitPastryMaster}) =$$
$$((\texttt{Liquid, Liquid})() \to \texttt{Liquid}()) \cap (\texttt{Pair}(\omega, \texttt{EggYolk}()) \to \texttt{EggYolk}()) \cap (\texttt{Pair}(\texttt{EggYolk}(), \omega) \to \texttt{EggYolk}())$$

Note that the Scala tuple `(Liquid, Liquid)` is mapped to a single sort and this sort is a single nullary type constructor, which is why the arity of `BiscuitPastryMaster` is 1 and there are additional empty parentheses in the intersection type. If we had any combinator using the Scala type `(Yolk, Yolk)`, there would be an entry `(Yolk, Yolk)` $\leq_{\mathbb{S}}$ `(Liquid, Liquid)` because Scala tuples are covariant. As target algebra we get $h_{\texttt{Liquid}}(S, \texttt{BiscuitPastryMaster}, x) = \texttt{BiscuitPastryMaster.apply}(x)$. All sorts are closed, so the space of well-formed substitutions only contains the empty substitution $S = \emptyset \in \{\emptyset\} = \emptyset \to \mathbb{S}(\emptyset) = \mathrm{WF}_1$ and h is substitution independent by definition. Semantic types however can include variables. In Listing 2 we defined a kinding

```
Kinding(kingdom.possession).addOption('Pieces('HeadShell)).addOption('Pieces('BottomShell))
  .merge(Kinding(kingdom.description).addOption('HeadShell).addOption('BottomShell))
```

that translates to $\mathrm{WF}_2 = \{\{\texttt{posession} \mapsto \texttt{Pieces(HeadShell)}, \texttt{description} \mapsto \texttt{HeadShell}\},$
$\{\texttt{posession} \mapsto \texttt{Pieces(BottomShell)}, \texttt{description} \mapsto \texttt{HeadShell}\},$
$\{\texttt{posession} \mapsto \texttt{Pieces(HeadShell)}, \texttt{description} \mapsto \texttt{BottomShell}\},$
$\{\texttt{posession} \mapsto \texttt{Pieces(BottomShell)}, \texttt{description} \mapsto \texttt{BottomShell}\}\}.$

To perform inhabitation, we use $\mathrm{WF} = \bigcup_{(S_1, S_2) \in \mathrm{WF}_1 \times \mathrm{WF}_2} \{S_1 \uplus S_2\} = \bigcup_{S_2 \in \mathrm{WF}_2} \{\emptyset \uplus S_2\} = \mathrm{WF}_2$ as well-formedness space. In a first step, this space is completely enumerated and all variables in Γ are replaced. In Listing 1 we have

$$\Gamma(\texttt{RoyalHotGlueAdvisor}) =$$
$$(\texttt{Seq[PreciousObject]}() \to \texttt{PreciousObject}()) \cap (\texttt{Pieces(description)} \to \texttt{description} \cap \texttt{Glued}())$$

which, after variable expansion by enumeration of substitutions, becomes an entry in the closed type context

$$\Gamma'(\texttt{RoyalHotGlueAdvisor}) = (\texttt{Pieces}(\texttt{BottomShell}()) \to \texttt{BottomShell}() \cap \texttt{Glued}()) \cap$$
$$(\texttt{Seq[PreciousObject]}() \to \texttt{PreciousObject}()) \cap (\texttt{Pieces}(\texttt{HeadShell}()) \to \texttt{HeadShell}() \cap \texttt{Glued}())$$

The inhabitation algorithm (Listing 5) iteratively builds up a tree grammar, starting from the empty grammar and a target. Tree grammars are represented by maps `TreeGrammar = Map[Type, Set[(String, Seq[Type])]]` with productions as entries. Each iteration step results in a new tree grammar and a stream of new recursive targets for every new entry in the new grammar. First (line 3) we check if there is a supertype of the current target, which is already known to be uninhabited. If this is the case, the current target must be uninhabited as well, so we may remove all right-hand sides which depend on it and return an empty stream of new recursive targets, indicating failure. Otherwise, we test if a type equal to the current target has already been tested. Then, we may substitute all dependencies on the current target by the known equal type and return a stream that includes one empty stream of new targets, indicating success without anything further to do. If the current target really is new, we organize it into paths (see Lem. 2.7). We then iterate (in parallel) over all (pre-organized) combinator types in the repository (line 9). We decompose each path of each organized combinator type into a vector of sources and a target, selecting just those decompositions for which there exists a path in the current inhabitation target that is a supertype of the decomposition target (line 10). This is sufficient because of the extended-path lemma 2.10. Different source counts cannot contribute to the same result, again because of the extended-path lemma, so we group path decompositions by source count (line 11). From each of these groups we select all possible pointwise intersections of all source vectors, such that intersecting their individually associated targets results in a type less than the current inhabitation target (line 12). After cancellation of duplicates and flattening (line 13) we get a map with a collection of pointwise intersected source vectors for every combinator. Application of a combinator to arguments typed by the associated source vectors yields a term that is typeable by the inhabitation target and the map contains all such possibilities for all combinators. As a final step, we create a new production for the grammar by rearranging the previously created map (line 15) and collect all new streams of recursive targets (line 18) from the source vectors. In the main iteration loop, which is not shown here for space reasons, we use that source vectors remain grouped when adding recursive targets (the inner streams returned by `inhabitStep`) and abort a whole group when one of its recursive targets fails. Iteration stops when no new inhabitation targets are found. As a final step, the tree grammar is pruned to remove all unproductive entries (e.g.,s $A ::= c(B), B ::= d(A)$). An excerpt of the grammar generated for the HumptyDumpty example looks like this:

`scala.Any` $\cap$ HumptyDumpty ::= FilledPastryMaster(`scala.Any` $\cap$ HumptyZombie, `Liquid` $\cap$ `EggYolk`) | *HumptyBeforeFall()*

`scala.Any` $\cap$ HumptyZombie ::= FilledPastryMaster(`scala.Any` $\cap$ `HeadShell` $\cap$ `BottomShell` $\cap$ Glued, `Liquid` $\cap$ `EggWhite`)

`Liquid` $\cap$ `EggYolk` ::= `BiscuitPastryMaster((Liquid, Liquid)` $\cap$ `Pair`(ω, `EggYolk`))

A word rooted in a nonterminal of the tree grammar is a combinatory expression with that type. In the example we have the semantic type `HumptyDumpty` and the native Scala type `scala.Any`. Words for these types start with FilledPastryMaster, which is the first combinator to apply. The two arguments are typed by `scala.Any` $\cap$ `HumptyZombie` and `Liquid` $\cap$ `EggYolk`, which both have grammar entries. If we had an additional operation, HumptyBeforeFall, which returns Humpty without any arguments, we would have the additional part (printed in *italicized text*) in our grammar, where the vertical bar | encodes a grammar alternative. Note, that the algorithm takes care of subtype equal nonterminals by replacing them with the first variation of the type that was found (line 5), hence equality of nonterminals may be considered to be syntactic equality.

Next, we need to enumerate words of the grammar to get inhabitants in form of applicative terms, one of which we have seen in listing 3. Tree grammars can be seen as polynomial functors. Duregård et al. (2012) describe FEAT (Functional Enumeration of Algebraic Datatypes), an efficient algorithm and a Haskell implementation to lazily enumerate such structures. A Scala port of FEAT is publicly available (Bessai 2016) and can directly be used.

Finally, before presentation to the user, each inhabitant is translated to a Scala value. Mathematically, this translation is the fixpoint family f from Lem. 2.21. In the implementation, we traverse the inhabitant and generate a quasi-quoted program text with calls to the apply method of each object (algebra h), and evaluate the runtime compiled code using Scala's `ToolBox.eval` as described in (Shabalin and Burmako 2017). Using disjointness (Lem. 2.14) we may split repository Γ into its native and semantic components, therefore the translation respects the sorts of the signature (the result is well-typed). The well-formed space WF is also split back into WF_1 and WF_2, where WF_1 only allows the empty substitution. Hence, results are unique, sound and complete for $\mathcal{AG}(\mathbb{D})(h)$, the fragment of Scala obtainable by calling apply methods of the specified objects.

```
1  @combinator object Score52 {
2    def apply(): Seq[Statement] = Java("if (getScoreValue() == 52) { return true; }").statements()
3    val semanticType:Type = 'WinConditionChecking :&: 'NonEmptySeq
4  }
5  class MoveWidgetToWidgetStatements(moveNameType:Type) {
6    def apply(rootPackage:NameExpr, theMove:NameExpr, movingWidget:NameExpr,
7              sourceWidget:NameExpr, targetWidget:NameExpr): Seq[Statement] =
8      controller.java.MoveWidgetToWidget.render(RootPackage = rootPackage, TheMove = theMove,
9                                 MovingWidget = movingWidget, SourceWidget = sourceWidget,
10                                TargetWidget = targetWidget).statements()
11   val semanticType:Type = 'RootPackage =>: 'MoveElement(moveNameType, 'ClassName) =>:
12     'MoveElement(moveNameType, 'MovableElement) =>: 'MoveElement(moveNameType, 'SourceWidget) =>:
13     'MoveElement(moveNameType, 'TargetWidget) =>: 'MoveWidget(moveNameType)
14 }
15 @combinator object ReserveToFoundationMove extends MoveWidgetToWidgetStatements('ReserveToFoundation)
16
17 abstract class AugmentCompilationUnit(base:Constructor, conceptType:Symbol) {
18   def fields(): Seq[FieldDeclaration];
19   def methods(): Seq[MethodDeclaration]
20   def apply(unit: CompilationUnit) : CompilationUnit = {
21     val types = unit.getTypes()
22     fields().foreach { x => types.get(0).getMembers().add(x) }
23     methods().foreach { x => types.get(0).getMembers().add(x) }
24     unit
25   }
26   val semanticType:Type = base =>: conceptType(base)
27 }
28 class GetterSetterMethods(att:NameExpr, attType:String, base:Constructor, conceptType:Symbol)
29          extends AugmentCompilationUnit(base, conceptType) {
30   def fields(): Seq[FieldDeclaration] =
31     Java(s"""protected $attType $att;""").classBodyDeclarations().map(_.asInstanceOf[FieldDeclaration])
32   def methods() : Seq[MethodDeclaration] = {
33     val capAtt = att.toString().capitalize
34     Java(s"""public $attType get$capAtt() { return this.$att; }
35              public void set$capAtt($attType $att) { this.$att = $att;}"""
36     ).classBodyDeclarations().map(_.asInstanceOf[MethodDeclaration])
37   }
38 }
39 @combinator object IncrementCombinator
40          extends GetterSetterMethods(Java("increment").nameExpression(), "int", 'Solitaire, 'increment)
```

Listing 6. Combinators building Java ASTs

4.2 Solitaire Product Line

A more realistic and complex application example is a product line of solitaire card games which takes advantage of a pre-existing Java framework, KombatSolitaire, that contains dozens of variations (Heineman et al. 2015). While Scala can interact with Java libraries, we choose to synthesize native Java code that extends this framework. The choice of Java as target language is arbitrary and to be taken as an example for one of the typical obstacles posed by external project requirements, such as pre-existing code, developer experience bias, or contractual agreements. We are, however, still able to apply the functional techniques and implementation which we have discussed so far. To this end, we synthesize ASTs of Java classes built by combinators written in Scala. Listing 6 demonstrates three such combinators. The first combinator, `Score52`, invokes a parser to construct a node in a Java AST from a string value. Semantically, the Java AST node represents the non-empty sequence of statements to determine the winning condition for a single-deck solitaire game. The constructed AST node is represented by `Seq[Statement]`, with `Statement` being part of the JavaParser library (van Bruggen et al. 2017). In the second combinator (line 15), the render method of a Twirl (Vlugter et al. 2017) template (line 8), shown in listing 7, is

```
1  @(RootPackage: NameExpr, TheMove: NameExpr, MovingWidget: NameExpr,
2    SourceWidget: NameExpr, TargetWidget: NameExpr)
3  /* Code is customized based on the source widget, the widget being dragged, the target widget,
4     and the designated move. These statements are eventually embedded within mouse release event handler. */
5  @Java(MovingWidget) movingElement = (@Java(MovingWidget)) w.getModelElement();
6  if (movingElement == null) { return; }
7
8  Widget sourceWidget = theGame.getContainer().getDragSource();
9  if (sourceWidget == null) { return; }
10 @Java(SourceWidget) source = (@Java(SourceWidget)) sourceWidget.getModelElement();
11 @Java(TargetWidget) toElement = (@Java(TargetWidget)) src.getModelElement();
12 // Construct actual move and attempt it
13 Move m = new @{Java(TheMove)}(source, movingElement, toElement);
14 if (m.valid(theGame)) {
15   m.doMove(theGame);
16   theGame.pushMove(m);
17 } else {
18   sourceWidget.returnWidget(w);
19 }
```

Listing 7. Twirl template building sequence of Java statements to request a move

Package	Abstract Class	Class	Combinators	Total	Java LOC	Time
generic	1	3		4		
shared		4	17	21		
FreeCell			117	117	1606	53 sec.
Idiot			14	14	675	21 sec.
Narcotic			14	14	661	23 sec.
Stalactites			97	97	1303	10 sec.
Grand Total	1	7	259	267	4265	107 sec.

Table 1. Table of combinators for solitaire product line

invoked on the combinator arguments. Within the template, we first declare all the parameters (lines 1 – 2) and then use them within the template body (e.g., line 5), pretty printing each parameter with `@Java()` directives.

The constructed code processes mouse release events to perform a valid move in a solitaire variation, which we again indicate using a semantic type. Combinator `IncrementCombinator` of listing 6 builds up a new AST from an existing one by inserting a field `increment` with getters and setters into a class. Such manipulation is commonplace, which is why the combinator code has been extracted into two classes (lines 17 and 28). This way, we can reuse the functionality in multiple combinators for different fields and methods without significant repetition. As indicated in Fig. 5, inhabitation targets are of type `CompilationUnit` (the AST Node type of full Java source files) and different semantic concepts for solitaire games. A small website serves the computed solutions and acts as presentation layer. Here, users can, by clicking on links, trigger computation of a solution (derivation of the inhabitant in tree form, translation and evaluation in Scala) and then will be presented with links to on-the-fly created Git repositories that contain the synthesized code. The Git repositories are internally hosted using JGit (Aniszczyk et al. 2017), a Java port of the Git versioning system.

To date the product line contains four synthesized variations implemented using 267 combinators, 25 of which are reused across different games (with 4 generic AST manipulation classes unspecific to card games). Table 1 shows a detailed account of the number of combinators involved in each game. The last two columns of Table 1 indicate the number of lines of synthesized Java code and the execution time for synthesis on a standard Windows PC with 8GB of RAM and an Intel i5 3.1 Ghz processor.

5 RELATED WORK

Lustig and Vardi (2009) prominently formulated the idea of synthesizing systems from component libraries, rather than from scratch. A large variety of technically different approaches have been proposed, ranging from co-algebraic automata theory (Bonsangue et al. 2008), graph- (Albarghouthi et al. 2013) and net-based (Feng et al. 2017) algorithms, to logic constraint solving (Gulwani et al. 2011). Rehof and Urzyczyn (2011) were the first to use type inhabitation in combinatory logic (Hindley and Seldin 2008) with intersection types (Barendregt et al. 1983). Study of the inhabitation question of intersection type systems, both for combinatory logic and lambda calculus gave rise to a series of theoretically (Bucciarelli et al. 2014; Düdder et al. 2012; Dudenhefner and Rehof 2017; Rehof 2013) and practically (Bessai et al. 2015a,b, 2014, 2016; Düdder et al. 2014; Heineman et al. 2016) significant results. All of these highlight the expressive power of intersection types, formalizing it using complexity theory and showcasing it by application to specific modeling tasks. Especially the idea of semantic types (Steffen et al. 1997), which is also used in adaptation synthesis via proof counting (Haack et al. 2002; Wells and Yakobowski 2004), can easily be translated to intersection types. Refinement types (Frankle et al. 2016; Freeman and Pfenning 1991) exemplify this for ML-style programs. Controlling enumerations with additional information is also a concern in FEAT (Duregård et al. 2012), where it can be addressed by boolean constraints (Claessen et al. 2014). Geared toward enumeration of data and limited to settings without polymorphism, FEAT is perfect for the enumeration of tree grammars constructed by the algorithm presented here. Similar to FEAT, the enumeration of non polymorphic Σ-Algebras is discussed by Burghardt (2002), but without an actual implementation. The approach presented in this paper improves upon prior results, especially for type-directed synthesis (Frankle et al. 2016; Osera and Zdancewic 2015; Polikarpova et al. 2016) and program completion (Gvero et al. 2013; Kuncak et al. 2010): based on well-known algebraic modeling techniques, it is highly expressive and target language agnostic. Being machine-checked and implemented, it also reaches a new level of formalization and practical applicability.

6 CONCLUSION AND FUTURE WORK

We have taken a long journey realizing a simple idea: with Σ-algebraic modeling techniques, it is possible to define and implement interfaces and then ask a type inhabitation algorithm for combinatory logic to produce all possible programs of a given type using only interface components. We started with an example in Scala (Sect. 1.1). We worked our way through the mathematics of subsorted $\Sigma_{\mathbb{S},\mathbb{V}}$-Algebras (Sect. 2.1), type checking and type inhabitation for combinatory logic with intersection types, distributing covariant constructors and finite well-formed substitution spaces (Sect. 2.2), to then connect the two worlds in a sound, complete and unique way (Sect. 2.3). We learned how to apply the resulting framework to Mini-ML$^{\Box}$ (Sect. 3), demonstrating its expressiveness and gains in generality over prior approaches to type inhabitation for staged systems (Düdder et al. 2014). Then we came full circle, showcasing an implementation in Scala (Sect. 4.1) with its connection to the initial example. We went further, evaluating applicability to synthesis of product line members of a family of solitaire games (Sect. 4.2). On our journey we remained truly language agnostic: just this paper shows application to the target languages of natural numbers, Mini-ML$^{\Box}$, Scala and to meta programming Java. We employed some of the Turing-complete modeling power (Rehof 2013) of semantic types to control composition, expressing concepts ranging from the filling of bagels, the placement of box-combinators to winning conditions in solitaire games – all of which are natural but difficult or impossible to express in target language type systems. Results are implemented and formalized in Coq and Scala. This especially includes the type inhabitation algorithm, which previously has been implemented (Düdder et al. 2014) and (in decision form) proven (Düdder et al. 2012), but had no machine-checked formalization. There is however a yet to address important part of algebraic modeling: signatures specifying relations of algebraic objects. This together with a Coq formalization of FEAT-like inhabitant enumeration and a formal finiteness proof of WF for Mini-ML$^{\Box}$ – which requires the tedious and hopefully automatable task of constructing a bijection into finite natural numbers – are potential areas of future work.

REFERENCES

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings.* 934–950. DOI: http://dx.doi.org/10.1007/978-3-642-39799-8_67

Chris Aniszczyk, Christian Halstrick, Colby Ranger, and others. 2017. JGit. (2017). https://eclipse.org/jgit/

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A Filter Lambda Model and the Completeness of Type Assignment. *J. Symb. Log.* 48, 4 (1983), 931–940. DOI: http://dx.doi.org/10.2307/2273659

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions.* Springer. DOI: http://dx.doi.org/10.1007/978-3-662-07964-5

Jan Bessai. 2016. Shapeless Generic Functional Enumeration of Algebraic Data Types for Scala. (2016). https://github.com/JanBessai/shapeless-feat

Jan Bessai, Boris Düdder, George T. Heineman, and Jakob Rehof. 2015a. Combinatory Synthesis of Classes Using Feature Grammars. In *Formal Aspects of Component Software - 12th International Conference, FACS 2015, Niterói, Brazil, October 14-16, 2015, Revised Selected Papers.* 123–140. DOI: http://dx.doi.org/10.1007/978-3-319-28934-2_7

Jan Bessai, Andrej Dudenhefner, Boris Düdder, Tzu-Chun Chen, Ugo de'Liguoro, and Jakob Rehof. 2015b. Mixin Composition Synthesis Based on Intersection Types. In *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland.* 76–91. DOI: http://dx.doi.org/10.4230/LIPIcs.TLCA.2015.76

Jan Bessai, Andrej Dudenhefner, Boris Düdder, Moritz Martens, and Jakob Rehof. 2014. Combinatory Logic Synthesizer. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I.* 26–40. DOI: http://dx.doi.org/10.1007/978-3-662-45234-9_3

Jan Bessai, Andrej Dudenhefner, Boris Düdder, Moritz Martens, and Jakob Rehof. 2016. Combinatory Process Synthesis. In *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I.* 266–281. DOI: http://dx.doi.org/10.1007/978-3-319-47166-2_19

Corrado Böhm and Alessandro Berarducci. 1985. Automatic Synthesis of Typed Lambda-Programs on Term Algebras. *Theor. Comput. Sci.* 39 (1985), 135–154. DOI: http://dx.doi.org/10.1016/0304-3975(85)90135-5

Marcello M. Bonsangue, Jan J. M. M. Rutten, and Alexandra Silva. 2008. Coalgebraic Logic and Synthesis of Mealy Machines. In *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings.* 231–245. DOI: http://dx.doi.org/10.1007/978-3-540-78499-9_17

Antonio Bucciarelli, Delia Kesner, and Simona Ronchi Della Rocca. 2014. The Inhabitation Problem for Non-idempotent Intersection Types. In *Theoretical Computer Science - 8th IFIP TC 1/WG 2.2 International Conference, TCS 2014, Rome, Italy, September 1-3, 2014. Proceedings.* 341–354. DOI: http://dx.doi.org/10.1007/978-3-662-44602-7_26

Jochen Burghardt. 2002. Axiomatization of Finite Algebras. In *KI 2002: Advances in Artificial Intelligence: 25th Annual German Conference on AI, KI 2002 Aachen, Germany, September 16–20, 2002 Proceedings*, Matthias Jarke, Gerhard Lakemeyer, and Jana Koehler (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 222–234. DOI: http://dx.doi.org/10.1007/3-540-45751-8_15

Koen Claessen, Jonas Duregård, and Michał H. Pałka. 2014. Generating Constrained Random Data with Uniform Distribution. In *Functional and Logic Programming: 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 18–34. DOI: http://dx.doi.org/10.1007/978-3-319-07151-0_2

Pierre-Louis Curien, Richard Garner, and Martin Hofmann. 2014. Revisiting the categorical interpretation of dependent type theory. *Theor. Comput. Sci.* 546 (2014), 99–119. DOI: http://dx.doi.org/10.1016/j.tcs.2014.03.003

Rowan Davies and Frank Pfenning. 2001. A modal analysis of staged computation. *Journal of the ACM (JACM)* 48, 3 (2001), 555–604. DOI: http://dx.doi.org/10.1145/382780.382785

Mariangiola Dezani-Ciancaglini and J. Roger Hindley. 1992. Intersection Types for Combinatory Logic. *Theor. Comput. Sci.* 100, 2 (1992), 303–324. DOI: http://dx.doi.org/10.1016/0304-3975(92)90306-Z

Boris Düdder, Moritz Martens, and Jakob Rehof. 2014. Staged Composition Synthesis. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings.* 67–86. DOI: http://dx.doi.org/10.1007/978-3-642-54833-8_5

Boris Düdder, Moritz Martens, Jakob Rehof, and Pawel Urzyczyn. 2012. Bounded Combinatory Logic. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France.* 243–258. DOI: http://dx.doi.org/10.4230/LIPIcs.CSL.2012.243

Andrej Dudenhefner, Moritz Martens, and Jakob Rehof. 2016. The Intersection Type Unification Problem. In *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, June 22-26, 2016, Porto, Portugal.* 19:1–19:16. DOI: http://dx.doi.org/10.4230/LIPIcs.FSCD.2016.19

Andrej Dudenhefner and Jakob Rehof. 2017. Intersection type calculi of bounded dimension. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017.* 653–665. DOI: http://dx.doi.org/10.1145/

3009837.3009862

Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. Feat: Functional Enumeration of Algebraic Types. In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. ACM, New York, NY, USA, 61–72. DOI:http://dx.doi.org/10.1145/2364506.2364515

Hartmut Ehrig and Bernd Mahr. 1985. *Fundamentals of Algebraic Specification 1: Equations und Initial Semantics*. EATCS Monographs on Theoretical Computer Science, Vol. 6. Springer. DOI:http://dx.doi.org/10.1007/978-3-642-69962-7

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-based synthesis for complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 599–612. DOI:http://dx.doi.org/10.1145/3009837.3009851

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-directed synthesis: a type-theoretic interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 802–815. DOI:http://dx.doi.org/10.1145/2837614.2837629

Timothy S. Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*. 268–277. DOI:http://dx.doi.org/10.1145/113445.113468

Martin Gogolla. 1984. Partially Ordered Sorts in Algebraic Specifications. In *Proc. Of the Conference on Ninth Colloquium on Trees in Algebra and Programming*. Cambridge University Press, New York, NY, USA, 139–153. http://dl.acm.org/citation.cfm?id=2868.2878

Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 62–73. DOI:http://dx.doi.org/10.1145/1993498.1993506

Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete Completion using Types and Weights. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 27–38. DOI:http://dx.doi.org/10.1145/2499370.2462192

Christian Haack, Brian Howard, Allen Stoughton, and Joe B. Wells. 2002. Fully Automatic Adaptation of Software Components Based on Semantic Specifications. In *Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, September 9-13, 2002, Proceedings*. 83–98. DOI:http://dx.doi.org/10.1007/3-540-45719-4_7

George T. Heineman, Jan Bessai, Boris Düdder, and Jakob Rehof. 2016. A Long and Winding Road Towards Modular Synthesis. In *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*. 303–317. DOI:http://dx.doi.org/10.1007/978-3-319-47166-2_21

George T. Heineman, Armend Hoxha, Boris Düdder, and Jakob Rehof. 2015. Towards migrating object-oriented frameworks to enable synthesis of product line members. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*. 56–60. DOI:http://dx.doi.org/10.1145/2791060.2791076

J. R. Hindley and J. P. Seldin. 2008. *Lambda-calculus and Combinators, an Introduction.* Cambridge University Press.

Hoare, C. A. R. 1972. Proof of correctness of data representations. *Acta Informatica* 1, 4 (1972), 271–281. DOI:http://dx.doi.org/10.1007/BF00289507

Martin Hofmann. 1997. Syntax and Semantics of Dependent Types. In *Semantics and Logics of Computation*, A. M. Pitts and P. Dybjer (Eds.). Vol. 14. Cambridge University Press, Cambridge, 79–130. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.8985

Bart Jacobs and Jan Rutten. 1997. A Tutorial on (Co)Algebras and (Co)Induction. *Bulletin of the European Association for Theoretical Computer Science* 62 (1997), 222–259. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.1418

Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. 2010. Complete functional synthesis. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*. 316–329. DOI:http://dx.doi.org/10.1145/1806596.1806632

K. Rustan M. Leino. 2015. Compiling Hilbert's epsilon operator. In *20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations, LPAR 2015, Suva, Fiji, November 24-28, 2015*. 106–118. http://www.easychair.org/publications/paper/Compiling_Hilberts_epsilon_operator

Yoad Lustig and Moshe Y. Vardi. 2009. Synthesis from Component Libraries. In *FOSSACS (LNCS)*, Vol. 5504. Springer, 395–409. DOI:http://dx.doi.org/10.1007/s10009-012-0236-z

Heather Miller, Eugene Burmako, and Philipp Haller. 2017. Scala reflection overview. (2017). http://docs.scala-lang.org/overviews/reflection/overview.html

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 619–630. DOI:http://dx.doi.org/10.1145/2737924.2738007

Peter Padawitz. 2011. From Grammars and Automata to Algebras and Coalgebras. In *Algebraic Informatics - 4th International Conference, CAI 2011, Linz, Austria, June 21-24, 2011. Proceedings*. 21–43. DOI:http://dx.doi.org/10.1007/978-3-642-21493-6_2

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 522–538.

DOI:http://dx.doi.org/10.1145/2908080.2908093

Jakob Rehof. 2013. Towards Combinatory Logic Synthesis. In *BEAT'13, 1st International Workshop on Behavioural Types, Held as Part of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2013 POPL'13, Rome, January 22 2013.* http://beat13.cs.aau.dk/pdf/BEAT13-proceedings.pdf

Jakob Rehof and Pawel Urzyczyn. 2011. Finite Combinatory Logic with Intersection Types. In *Typed Lambda Calculi and Applications - 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings.* 169–183. DOI:http://dx.doi.org/10.1007/978-3-642-21691-6_15

Robert AG Seely. 1984. Locally cartesian closed categories and type theory. In *Mathematical proceedings of the Cambridge philosophical society*, Vol. 95. Cambridge Univ Press, 33–48. DOI:http://dx.doi.org/10.1017/S0305004100061284

Denys Shabalin and Eugene Burmako. 2017. Scala Quasiquotes. (2017). http://docs.scala-lang.org/overviews/quasiquotes/intro.html

Srinivas, Yellamraju V. and Jüllig, Richard. 1995. Specware: Formal support for composing software. In *Mathematics of Program Construction: Third International Conference, MPC '95 Kloster Irsee, Germany, July 17–21, 1995 Proceedings*, Möller, Bernhard (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 399–422. DOI:http://dx.doi.org/10.1007/3-540-60117-1_22

Bernhard Steffen, Tiziana Margaria, and Michael von der Beeck. 1997. Automatic Synthesis of Linear Process Models from Temporal Constraints: An Incremental Approach. In *In ACM/SIGPLAN Int. Workshop on Automated Analysis of Software (AAS'97).* http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.25.8319

The Coq development team. 2016. *The Coq proof assistant standard library.* LogiCal Project. https://coq.inria.fr/distrib/8.5pl2/stdlib/ Version 8.5p2.

Danny van Bruggen and others. 2017. JavaParser. (2017). http://javaparser.org

Peter Vlugter and others. 2017. Twirl - The Play Scala Template Compiler. (2017). https://github.com/playframework/twirl

J. B. Wells and Boris Yakobowski. 2004. Graph-Based Proof Counting and Enumeration with Applications for Program Fragment Synthesis. In *Logic Based Program Synthesis and Transformation, 14th International Symposium, LOPSTR 2004, Verona, Italy, August 26-28, 2004, Revised Selected Papers.* 262–277. DOI:http://dx.doi.org/10.1007/11506676_17

Martin Wirsing. 1986. Structured Algebraic Specifications: A Kernel Language. *Theor. Comput. Sci.* 42 (1986), 123–249. DOI:http://dx.doi.org/10.1016/0304-3975(86)90051-4

Received February 2017